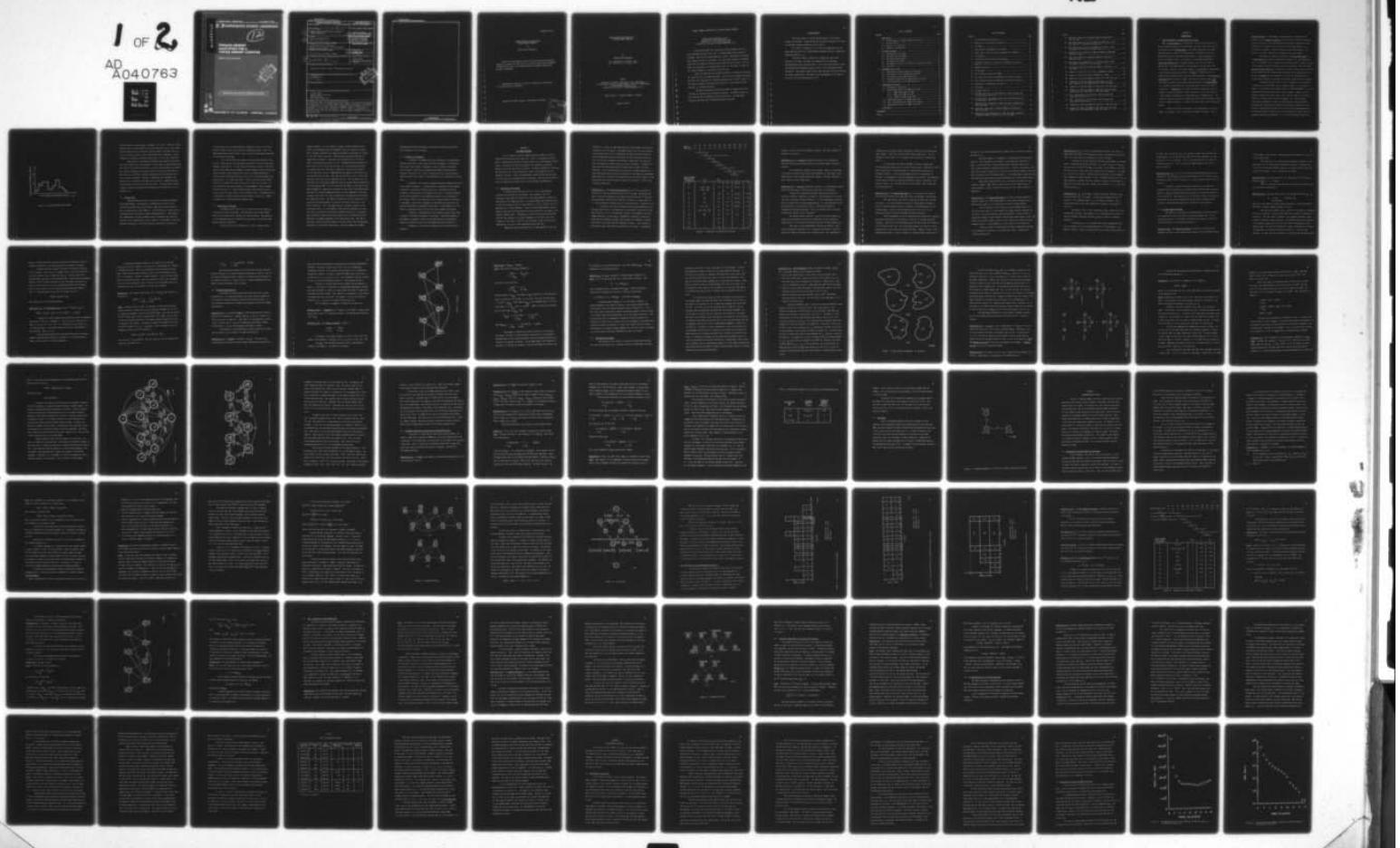
AD-A040 763 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
DYNAMIC MEMORY ALLOCATION FOR A VIRTUAL MEMORY COMPUTER.(U)
JAN 77 R L BUDZINSKI DAAB07-72-C-0259
UNCLASSIFIED R-754 NL

1 OF 2
AD A040763



REPORT R-754 JANUARY, 1977 UILU-ENG 77-2201

CSL COORDINATED SCIENCE LABORATORY

12 B.S.

DYNAMIC MEMORY ALLOCATION FOR A VIRTUAL MEMORY COMPUTER

ROBERT LUCIUS BUDZINSKI

UNIVERSITY OF ILLINOIS - URBANA, ILLINOIS

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

DYNAMIC MEMORY ALLOCATION FOR A VIRTUAL MEMORY COMPUTER.

5. TYPE OF REPORT & PERIOD COVERED

Technical Report

6. PERFORMING ORG. REPORT NUMBER

R-754, UILU-ENG-77-2201

7. AUTHOR(s)

Robert Lucius Budzinski

8. CONTRACT OR GRANT NUMBER(s)

DAAB-07-72-C-0259;
NSF-MCS-73-03488-A01

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS

Joint Services Electronics Program

12. REPORT DATE

January, 1977

13. NUMBER OF PAGES

140

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

Doctoral thesis,

15. SECURITY CLASS. (of this report)

UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

149 p.

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Virtual Memory
Dynamic Memory Allocation
Multiprocessing
Space-Time Memory Cost

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

An algorithm, DMIN, for computing an optimal dynamic allocation which minimizes the space-time cost in a demand paging virual memory is presented. The problem is represented by a graph. We present an algorithm which reduces the size of the graph. Its worst case complexity is $O(DP*N^i)$ where DP is the number of distinct pages referenced in the trace. N is the number of references in a processed trace, and i is the number of nodes in largest subgraph to be removed at any one time.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

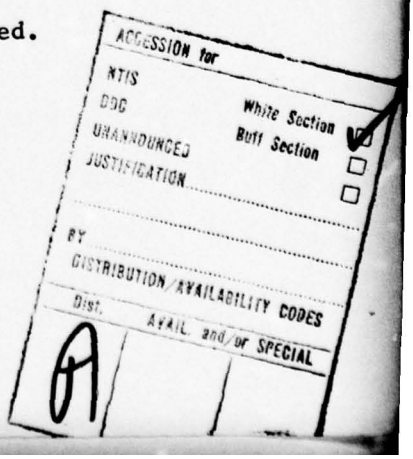
UILU-ENG 77-2201

DYNAMIC MEMORY ALLOCATION FOR A VIRTUAL MEMORY COMPUTER

by

Robert Lucius Budzinski

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259 and in part by the National Science Foundation under Grant NSF MCS 73-03488-A01.



DYNAMIC MEMORY ALLOCATION FOR A VIRTUAL MEMORY COMPUTER

BY

ROBERT LUCIUS BUDZINSKI

B.S., University of Illinois, 1972
M.S., University of Illinois, 1974

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1977

Thesis Advisor: Professor Edward S. Davidson

Urbana, Illinois

DYNAMIC MEMORY ALLOCATION FOR A VIRTUAL MEMORY COMPUTER

Robert Lucius Budzinski, Ph.D.
Coordinated Science Laboratory and
Department of Electrical Engineering
University of Illinois at Urbana-Champaign, 1977

An algorithm, DMIN, for computing an optimal dynamic allocation which minimizes the space-time cost in a demand paging virtual memory is presented. The problem is represented by a graph. We present an algorithm which reduces the size of the graph. Its worst case complexity is $\mathcal{O}(DP*N^{i})$ where DP is the number of distinct pages referenced in the trace. N is the number of references in a processed trace, and i is the number of nodes in the largest subgraph to be removed at any one time.

After the reduction, the optimal allocation is found by determining the maximum flow in the remaining graph. For the class of graphs considered, the worst case complexity for finding the maximum flow is $\mathcal{O}(N'^{3})$ where N' is the number of nodes in the graph after reduction. We describe an implementation of the DMIN allocation.

The memory allocation resulting from DMIN is compared with that from MIN, an optimal static allocation algorithm. The DMIN allocation is also compared with two dynamic heuristic algorithms: the Page Fault Frequency algorithm and the Damped Working Set algorithm.

ACKNOWLEDGMENT

The author wishes to express deep gratitude to his advisor, Professor Ed Davidson. Professor Davidson's guidance has made this thesis an invaluable learning experience for the author.

The author is indebted to H. S. Stone for suggesting the flow graph characterization for finding minimum subgraphs and to W. Mayeda for suggesting Theorem 3.1.1.

The author wishes to thank his colleagues for their help, especially Joel Emer, Alan Gant, Dan Hammerstrom, Bill Kaminsky, Balasubramanian Kumar, and Janak Patel. Also, the author wishes to thank the drafting and photography support groups of the Coordinated Science Laboratory. The typing of Hazel Corray, Mary McMillen, and Trudy Williams are greatly appreciated. The support and encouragement of the author's wife, Lyn, has been indispensable to him.

TABLE OF CONTENTS

LIST OF FIGURES

Figure | Page

CHAPTER 1

INTRODUCTION

1.1. Optimum Paging for Dynamic Memory Allocation

A virtual memory is a hierarchial or multi-level memory with an address mapping mechanism and an allocation algorithm. The first level of the hierarchy is the primary memory. The primary memory has the highest speed and smallest capacity in the hierarchy. Subsequent levels have decreasing speed and increasing storage capacity. For our purposes, we call all subsequent levels, collectively, the secondary memory. The addressing mechanism maps a program's address space into the physical memory space. The address space of the program can be much larger than the available capacity of the primary memory. For this thesis, ~~we consider~~ was employed. a page-oriented addressing mechanism. A page is a fixed size block of computer words associated with contiguous memory addresses. The allocation algorithm manages the flow of pages to and from the primary memory. For this paper we assume the allocation algorithm employs only demand paging, i.e., a page is transferred into the primary memory only as the result of a page fault. A page fault occurs when the program references a page not resident in the primary memory. When a page fault occurs, the execution of the program is suspended until the needed page is transferred in. The allocation algorithm attempts to manage page flow so that the average access time for a memory reference is close to the access time of the fast primary memory.

There are two basic strategies for allocating space in primary memory to a program: static allocation and dynamic allocation. With a

static allocation, a fixed number of memory pages is allocated to the program. With a dynamic allocation, the size of the allocation is allowed to vary according to the current needs of the program. The goal of both strategies is to maximize the utilization of the primary memory, i.e., to maximize the number of memory references in any time interval. This objective is roughly equivalent to maximizing the job throughput.

One measure of memory utilization is the space-time product [1] or space-time cost of memory used during a program's run. The space-time cost for a program's run is defined as the average amount of memory allocated to the program multiplied by the run time of the program. For this thesis, the unit of space-time cost is defined as one page of memory space multiplied by the average time per primary memory reference when no page faults occur. We use the average time per reference to account for the possibility of primary memory reference interference and uneven request rate by a program.

Minimizing the space-time cost for a program's run is related to increasing the job throughput for the computer system, as illustrated in Figure 1. We assume that the primary memory is "multiprogrammed," i.e., the primary memory may be shared by two or more programs. Figure 1 shows a program's dynamic allocation during a run. The area under the allocation profile is the space-time cost for a program's run. Other programs concurrently scheduled can share the remaining space with the maximum allocated space being M pages (physical primary memory available in the system). By minimizing the space-time cost for each program separately, the maximum number of programs that can be scheduled in a time interval is increased under ideal scheduling, i.e., jobs are scheduled so that the

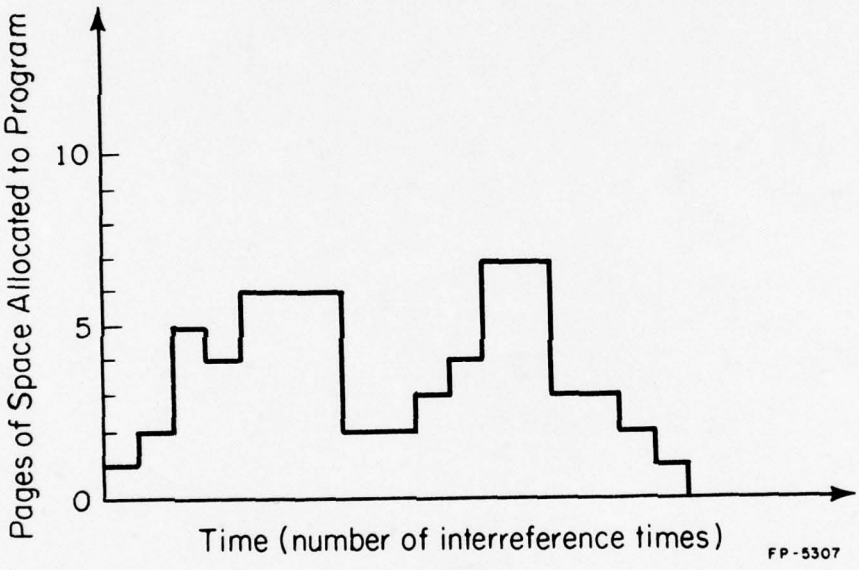


Figure 1. A typical memory space profile.

allocated space is exactly equal to M pages at all times. Since the system provides M units of space-time in each inter-reference time, if the average program takes k units of space-time, then the number of programs that can be run in T inter-reference times is MT/k under ideal scheduling. System throughput is therefore inversely proportional to k with ideal scheduling.

Realistically, individually minimizing each program's space-time area (cost) implies that more programs' profiles might be packed (scheduled) into the M x T space-time plane which in turn implies increased job throughput. Maximum throughput occurs when jobs with allocations that minimize space-time cost are ideally scheduled (assuming that the available jobs can be ideally scheduled). The goal of maximizing job throughput thus separates into a problem of job space allocation and a scheduling problem. In this thesis we attack the allocation problem. We present and evaluate a dynamic allocation algorithm which minimizes the space-time cost for a program's run.

1.2. Previous Work

There has been much work in the area of allocation algorithms using demand paging. Many heuristic (implementable on line), static allocation algorithms have been studied. Among them are Last-In-First-Out, and Least Recently Used (LRU) [2]. The performance of these heuristic algorithms has been bounded by Belady's MIN algorithm [3]. MIN requires prior knowledge of the page reference trace to determine the allocation, and thus is not implementable on line. However, by using MIN to bound the heuristic algorithms, it has been found that the number of page faults

occurring when using the LRU algorithm is generally close to the minimum possible number of page faults (found by the MIN algorithm). Thus, the MIN algorithm made further work on static allocation algorithms unnecessary for present day technology.

Also, several heuristic dynamic allocation algorithms have been studied. Among them are the Working Set algorithm [4,5] and the Page Fault Frequency algorithm [6]. In this thesis, we develop a bound on the performance of these algorithms with DMIN, an optimal dynamic allocation algorithm. Given a program's page reference trace from a run, DMIN determines an allocation which minimizes the space-time cost for the primary memory used during the run. We assume that there is always sufficient primary memory available to contain the optimal allocation for each active program. We also assume that the primary memory is multiprogrammed to make a dynamic allocation sensible. A single program running alone would obviously be given a static allocation equal to the available memory. The DMIN algorithm is applicable to computer systems with any number of processors. DMIN is also applicable to any hierarchical memory system.

1.3. Objectives of this Work

The DMIN algorithm can be used to settle some of the questions concerning allocation algorithms. The performance gap between dynamic heuristic algorithms and the optimum can now be evaluated. The magnitude of this performance gap will determine the usefulness of developing other dynamic heuristic algorithms.

Another issue is the comparison of a static strategy versus a

dynamic strategy. In a real system, a dynamic strategy requires a more complex job scheduling algorithm compared to one for a static allocation. Thus, a dynamic allocation must improve performance sufficiently to justify its use over static allocation. There are two performance measures for judging both strategies: space-time cost accumulated for a run and the number of page faults occurring in a run. The MIN algorithm minimizes the space-time cost for a run given the size of its fixed allocation because it minimizes run time. Run time is composed of execution time and wait time for page fault service. MIN minimizes the number of page faults, and thus minimizes the run time. Thus since the allocation size is fixed, MIN minimizes space-time cost by minimizing run time. For the same trace, the space-time cost from applying DMIN will never be greater than the space-time cost from applying MIN. It is not known how the number of page faults from applying DMIN will compare with the number of page faults from applying MIN. The results of comparing DMIN with MIN are strongly dependent on the page reference traces used, but there are additional variables which will affect the comparison results. The number of page faults and the space-time cost resulting from applying MIN to a trace are functions of the size of the static allocation. Also, the space-time cost increases linearly as the reactivation time, i.e., the length of time from the occurrence of a page fault until the program is reactivated (resumes execution), is increased. Both the number of page faults and the space-time cost resulting from the application of DMIN are dependent on the reactivation time. The size of the static allocation and the length of the reactivation time are dependent upon aspects of the system architecture. Thus the comparison of DMIN's

performance with MIN's performance will reveal how architecture affects the performance of each strategy.

1.4. Summary of the Research

In Chapter 2 the DMIN algorithm is developed. The nonreference interval model and the space-time cost model are defined. We represent these models with a graph. The graph is used to define a flow-graph. The flow-graph is then used to determine the dynamic allocation which minimizes space-time cost. The set of page faults for one reactivation time is proven to be a subset of the page faults for a smaller reactivation time.

In Chapter 3, we develop a subgraph reduction technique applicable to the graph of Chapter 2. A related graph is developed from the methods of Chapter 2 using a dual starting allocation. The general reduction technique is then applied to both graphs concurrently. Disjoint subgraphs of the reduced graph are treated separately without loss of generality. The savings in worst case complexity to the flow-graph method of Chapter 2 is evaluated. Some faster techniques for evaluating the space-time cost bound are developed for suboptimal allocations. An implementation of the reduction technique is applied to the two coupled graphs concurrently.

In Chapter 4, experimentation with computer program traces under a variety of allocation strategies is performed. Allocations resulting from DMIN are compared to those from the MIN algorithm, the Page Fault Frequency algorithm, and also from the Damped Working Set algorithm.

In Chapter 5, conclusions and suggestions for future work are presented.

CHAPTER 2

THE DMIN ALGORITHM

In this chapter we develop DMIN, an optimal dynamic allocation algorithm which minimizes space-time cost. First, a nonreference interval model is developed to describe a program running on a virtual memory computer system. We develop the relevant properties of nonreference intervals. Second, we form the space-time cost model. We present a method for computing the space-time cost of an allocation in terms of the properties of nonreference intervals. Third, the nonreference interval model and the space-time cost model are represented with a flow-graph. The optimal allocation is found by computing the maximum flow in the flow-graph representation.

2.1. Development of the Model

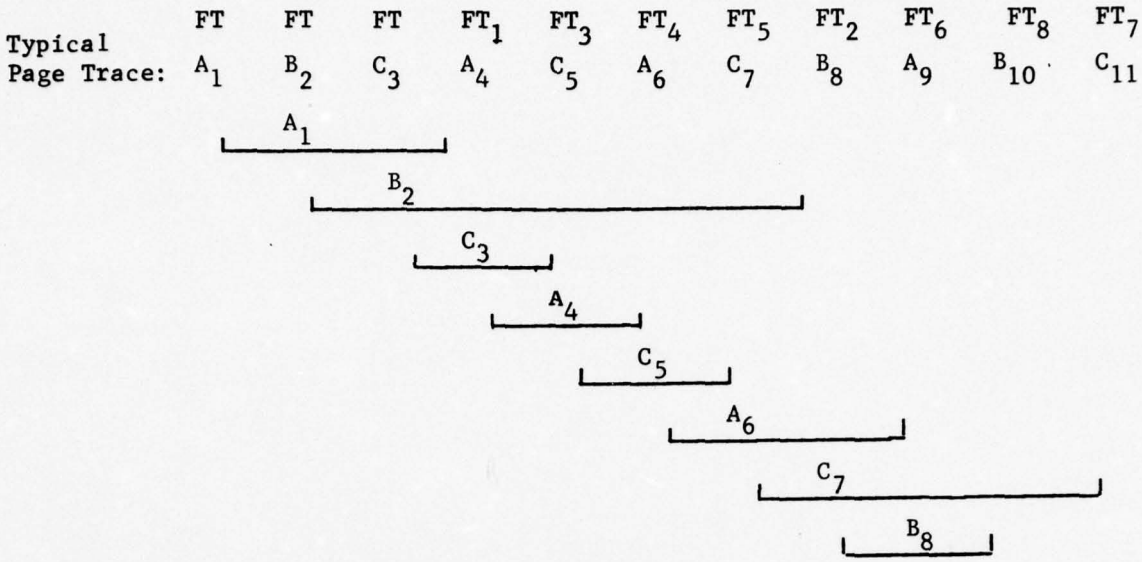
A program's page reference trace is a list of page numbers consecutively referenced by the program. Each element in the list represents a page that must either be in the primary memory or brought in. The problem of the DMIN algorithm is to decide whether each referenced page is left in the memory until its next reference or will be brought in via a page fault at its next reference. Since we assume demand paging, there is always a page fault associated with the first reference to each page in the program's address space. Furthermore, the allocation decision associated with the last reference to a page is degenerate since there is no next reference to that page. After its last reference, a page is deallocated immediately so as not to incur unnecessary space-time cost.

Immediately after each reference to a page (except for the last

reference to a page) the DMIN algorithm must decide whether to deallocate the page or leave the page in the primary memory until the next reference. If the optimal decision is to deallocate the page, the decision must be executed immediately after the reference in order to minimize space-time cost. If the next reference occurs without a page fault, the page must have been resident in memory since its previous reference. Otherwise the page would have been deallocated and brought back in before its next reference. Such a procedure would be contrary to a demand paging policy. Thus, (excluding the first and last reference to each page in the program's address space) an allocation or deallocation decision must be made for each page during every interval between consecutive references. Consider the following definitions:

Definition 2.1.1 The Nonreference Interval of the i^{th} page reference is the period of time starting just after the i^{th} reference and ending just prior to the next reference to the page associated with the i^{th} reference. The nonreference interval before the first reference to a page starts at $-\infty$. The nonreference interval after the last reference to a page ends at ∞.

In Figure 2, we have illustrated a typical page reference trace. The page trace starts at the beginning of the program's execution and ends when execution is over. Nonreference interval A_1 is spanned by the line segment labeled A_1 in the figure. Nonreference interval A_1 starts just after the last reference to page A and includes all references to pages B and C which occur before the next reference to page A. Also interval A_1 includes any page faults which occur during this interval. Nonreference



TRACE ELEMENT (Nonreference Interval)	$\lvert F_i \rvert$	USE	τ_i	P_i	Δ_i
A_1	2	R	2.1R	$\{B_2, C_3\}$	-.1R
B_2	$1 + FT_1 + FT_3 + FT_4 + FT_5$	2R	1.9R	$\{A_6, C_7\}$	1.1R
C_3	FT_1	.1R	.1R	$\{A_4, B_2\}$	3.9R
A_4	FT_3	.1R	.1R	$\{B_2, C_5\}$	3.9R
C_5	FT_4	.1R	1.1R	$\{A_6, B_2\}$	2.9R
A_6	$FT_5 + FT_2$	1.1R	1.1R	$\{B_8, C_7\}$	2.9R
C_7	$FT_2 + FT_6 + FT_8$	.5R	1.8R	ϕ	.2R
B_8	FT_6	.6R	1.1R	$\{C_7\}$	1.9R
A_9	–	1.1R	∞	–	$-\infty$
B_{10}	–	.1R	∞	–	$-\infty$
C_{11}	–	2R	∞	–	$-\infty$

Figure 2. Properties of nonreference intervals

interval A_1 ends at the next reference to page A. This next reference is labeled A_4 in Figure 2.

Definition 2.1.2 An Occupied nonreference interval is a nonreference interval whose associated page remains in the primary memory during the interval.

Let nonreference interval A_1 be occupied. Then, A_1 is resident in the memory during all page faults and references to other pages in the address space of the program during this interval. The next reference to page A does not cause a page fault.

Definition 2.1.3 A Vacated nonreference interval is a nonreference interval whose associated page is deallocated at the start of the interval.

Consider the nonreference interval which ends at the first reference to page B, labeled B_2 in Figure 2. From Definition 2.1, we see that this nonreference interval starts at $-\infty$. This nonreference interval must be vacated since we assume demand paging, i.e., this page must not be allocated space in the memory at time $-\infty$. Otherwise we would be implementing a prepaging strategy. The first reference after a vacated nonreference interval causes a page fault.

The optimal dynamic allocation problem can now be restated as: assign each nonreference interval of a given trace to the occupied or vacated state so as to minimize the space-time cost for that program run.

The states of some nonreference intervals are obvious. Zero-length nonreference intervals (i.e., consecutive references to the same page) will always be assigned to the occupied state. For convenience of

computation, we lump zero-length nonreference intervals into a single page trace element. With this action we form a reduced page trace. For the remainder of this thesis, it is assumed that we are using a "reduced page trace."

For the purpose of developing DMIN, we assume a program with allocated primary memory space is in one of three states: executing, blocked while a page fault is being processed, or blocked while waiting for an available processor. For developing DMIN, there are two parameters of interest. One parameter is entirely dependent on the system. This parameter is the length of time from leaving the execution state to returning to the execution state, i.e., the reactivation of the program. Thus we make the following definition:

Definition 2.1.4 The Reactivation Time, R, of a program is the length of time from the occurrence of a page fault until the program is reactivated.

The reactivation time interval typically includes the following activities: call to the operating system for virtual memory overhead, a wait for secondary memory service, a transfer of the needed page into the primary memory, a wait for an available processor, and a call to the operating system to initialize the CPU and restart execution.

With the parameter, R, we can characterize a broad range of systems. The memory architecture of the system is reflected in the page retrieval component. The levels of multiprogramming and multiprocessing are reflected in the processor wait time. For our present purposes, we choose to use an average value of R in order to simplify the model. However, in more complex sophisticated models, R could be modeled as a random variable selected as a

function of other system parameters and present levels of activity, queuing, allocation, etc.

The other parameter is dependent on system properties and program behavior. This parameter is the length of time required to deallocate a page. A page which has not been modified during execution requires very little system overhead to complete deallocation. In such a circumstance, the space that the page occupies can be written over because we assume an identical copy is available in the secondary memory. On the other hand, a page which has been modified during execution must be transferred out of the primary memory.

Deallocation begins with a call to the operating system for page transfer overhead. Next, there is wait time for the secondary memory to become available. Finally, there is the transfer time. Thus we make this definition:

Definition 2.1.5 The Deallocation Time, D, is either the time required to transfer a page out of the primary memory or the system overhead time for deallocating a page if it is unnecessary to perform a page transfer.

For the purposes of this thesis, we use an average value of D. The deallocation time, D, can be modeled as a random variable dependent on the state of the system and the system configuration. In this thesis, we assume that whenever a page transfer involving the primary memory occurs, the space in the primary memory involved in the transfer is occupied during the transfer process. In the examples, D is set equal to R for simplicity.

Now we can begin to characterize the useful properties associated with nonreference intervals (or "intervals" for short). Consider the following definitions:

Definition 2.1.6 F_i is the set of nonreference intervals that result in page faults during the i^{th} nonreference interval, including the page faults that occur due to the first reference to a particular page.

In Figure 2, we have placed an "FT" above each element in the page trace. A subscripted FT, FT_j, has value 1 if the j^{th} interval is vacated. FT_j has value 0 if the j^{th} interval is occupied. An FT without a subscript has value 1 since a fault must occur before the first reference to the page at the end of the interval. In the column labeled $|F_i|$, we have computed the cardinality of each interval's F set. Consider interval B_2's F set, F_2. $|F_2|$ contains a 1 because the first reference to page C occurs within B_2. $|F_2|$ has the terms FT_1, FT_3, FT_4, and FT_5 since intervals 1, 3, 4, and 5 end within interval B_2.

Definition 2.1.7 τ_i is the number of memory references (execution time) occurring within the i^{th} interval. If the page associated with the i^{th} interval is not referenced again, τ_i is set equal to infinity. For an interval that ends at the first reference to a page, τ_i is set equal to infinity.

In Figure 2, the τ's are calculated as sums of USE times in the corresponding interval. For example, τ_1 is 2.1R because the B_2 entry is used for 2R memory references and the C_3 entry is used for .1R memory references.

In Definition 2.1.7, an interval that begins immediately after the last reference to its associated page has its corresponding τ set equal to infinity. This is consistent with the fact that such intervals do

not end. Also, by setting such τ_i to infinity, we mark such intervals for being vacated. For an interval that ends at the first reference to a page, we also set its τ_i to infinity. This is consistent with the fact that the interval starts at $-\infty$. Also, since we use demand paging, this interval must be vacated.

Definition 2.1.8 P_i is the set of occupied nonreference intervals with two properties: these intervals start before the end of the i^{th} interval, and end after the end of the i^{th} interval, i.e., they overlap the end of the i^{th} interval.

In Figure 2, we have listed each interval's P_i set under the assumption that all intervals are occupied except infinite length intervals. Thus P_1 is $\{B_2, C_3\}$ because intervals B_2 and C_3 are occupied and overlap the end of interval A_1. The elements of P_1 can be readily seen in the figure by noting that the B_2 interval segment and the C_3 interval segment would be crossed by a line drawn perpendicular to the end of the A_1 interval.

2.2. The Space-Time Cost Model

Our approach to attacking the optimal dynamic allocation problem is to begin with an allocation which could be the optimal one. If this starting allocation is not optimal, we modify the allocation to reach the optimal allocation. Our starting allocation is:

Definition 2.2.1 The Starting Allocation is formed by setting all non-reference intervals to the occupied state, except for those intervals whose

corresponding τ's are infinite. Those intervals with infinite τ's are set to the vacated state.

The space-time cost of the starting allocation is related to the space-time cost for each occupied nonreference interval. The space-time cost for an occupied nonreference interval is one page times the real time length of the interval. Thus, consider the following:

Definition 2.2.2 The space-time cost for occupying the i^{th} nonreference interval, CIN_i, is: $\tau_i + |F_i|*R$.

This cost is equal to the execution time within the interval plus the time spent on page faults.

Definition 2.2.3 The space-time cost for the starting allocation, C_s, is:

$$C_s = \sum_{\forall i | i \text{ occupied}} CIN_i + DP*(R+D) + MR ,$$

where DP is the number of distinct pages referenced in the trace and MR is the number of memory references made in the run.

The starting allocation cost is composed of two types of components. One type of component, the sum of the CIN's, can be altered by changing the allocation. The other type of component, the remaining two terms, arises because of computer system constraints. The DP*(R+D) term is the space-time cost assumed for page traffic for the first and last references to each distinct page. For the first reference to a page, we assume that the space for that page is occupied as the page is brought in. After the last reference to a page, we assume the space is occupied during the deallocation

process. The MR term arises because in defining a nonreference interval, the memory references to the page preceding the interval is excluded.

Suppose we decide to modify the initial allocation by vacating the i^{th} interval. Then, how is C_s changed? First, since this interval is no longer occupied, CIN_i is subtracted from C_s. Secondly, some of the remaining CIN's that form C_s have been modified. Every interval overlapping the end of the i^{th} interval has had its real time length increased by R. Thus, for each interval in P_i, the corresponding CIN must be increased. Finally, we must add the cost for deallocating the page and bringing it back in. Combining terms we have that the change in cost is:

$$((D+R) + |P_i|*R) - CIN_i \ .$$

This leads us to the following definition:

Definition 2.2.4 The Differential Cost for the i^{th} interval, Δ_i, is:

$$((D+R) + |P_i|*R) - CIN_i = D + R + |P_i|*R - \tau_i - |F_i|*R \ .$$

In Figure 2, we have computed each interval's Δ with the assumption that D has a value of R. Consider Δ_1. $|P_1|$ is 2; τ_1 is 2.1R; $|F_1|$ is 2. Thus $\Delta_1 = R + R + 2R - 2.1R - 2R = -.1R$. Similarly for Δ_2, $|P_2|$is 2; τ_2 is 1.9R; and $|F_2|$ is 1 in the starting allocation (all subscripted FT's have value 0). Thus $\Delta_2 = 1.1R$.

From the above discussion we see that the value of Δ influences whether a page should be vacated or occupied. If Δ is less than zero, then it is less costly to vacate the interval.

When vacating several intervals, the change in cost from the starting allocation is related to the sum of the corresponding Δ's. However, a complication arises. The Δ's are computed with the starting allocation made. When vacating several intervals, some of the corresponding Δ's may have to be modified. Consider a vacated interval i. The P_i set was formed with the starting allocation. But if an interval in the P_i set is vacated, the P_i set must be modified. Thus, consider a set of intervals that are to be vacated, and call this an OUT set. We have the following:

Theorem 2.2.1 The change in space-time cost for vacating the intervals in an OUT set is

$$C_{\Delta}(OUT) = \sum_{i \in OUT} \Delta_i - \sum_{i \in OUT} | P_i \cap OUT | * R$$

Proof: Consider interval $k \in$ OUT. By changing its state from occupied to vacated, we need to replace its cost for occupying the memory to its cost for vacating the memory. Thus we must subtract CIN_k. The cost for vacating the k^{th} interval must include the page traffic cost, (D+R), plus the space occupied by the pages which are in the memory during the page service time at the end of the interval. The number of pages in the memory at the end of the k^{th} interval is $| P_k | - | P_k \cap OUT |$. Thus the change in cost for changing the state of the k^{th} interval is

$$(D+R) + | P_k | * R - | P_k \cap OUT | * R - CIN_k ,$$

but this is $\Delta_k - | P_k \cap OUT | * R$. Thus the change in cost for vacating the intervals in an OUT set is:

$$\sum_{i \in OUT} \Delta_i \quad - \quad \sum_{i \in OUT} | P_i \cap OUT | * R \quad = \quad C_\Delta(OUT)$$

□

The problem that remains is to find the OUT set that causes the greatest decrease in cost when vacating the intervals in the set. This is an integer programming problem. However, by transforming the problem, we can reduce the problem to a special class of integer programming problems, namely finding the maximum flow in a graph.

2.3. The Graph Representation

The optimal allocation problem maps very nicely into a graph representation. The graph representation provides physical insight into the interaction among nonreference intervals which become vacated. We first define some properties of graphs, and secondly we develop the graph representation.

Definition 2.3.1 An undirected graph, G, with weighted nodes and edges is defined by an ordered pair: $[N_G, E_G]$, where N_G is the set of nodes of G, and E_G is the set of edges of G. Each edge is an unordered pair of nodes in N_G. For each node, $i \in N_G$, the corresponding node weight is NW(i). For each edge, $e \in E_G$, the corresponding edge weight is EW(e).

The properties of the nonreference interval model and the space-time cost model can be included in a graph representation.

Definition 2.3.2 A G-graph is a graph $G = [N_G, E_G]$. The nodes of N_G correspond one-to-one with the finite length nonreference intervals of a

given trace (the last reference to each page has an infinite nonreference interval). The weight of each node is the Δ for the corresponding nonreference interval. Each node has the same name as its corresponding nonreference interval. An edge is connected between node A_i and node B_j if and only if $A_i \in P_j$ or $B_j \in P_i$, i.e., one nonreference interval overlaps the end of the other nonreference interval. Every edge has weight R.

In Figure 3, we have constructed a G-graph from the example of Figure 2. Each node is labeled by its corresponding nonreference interval. Each edge has weight R. There is an edge between each pair of nodes whose associated intervals overlap. This condition is equivalent to one interval overlapping the end of the other. For example, A_1 and C_3 are connected because $C_3 \in P_1$. B_2 and A_6 are connected because $A_6 \in P_2$.

For our purposes, a subgraph is defined by its nodes only.

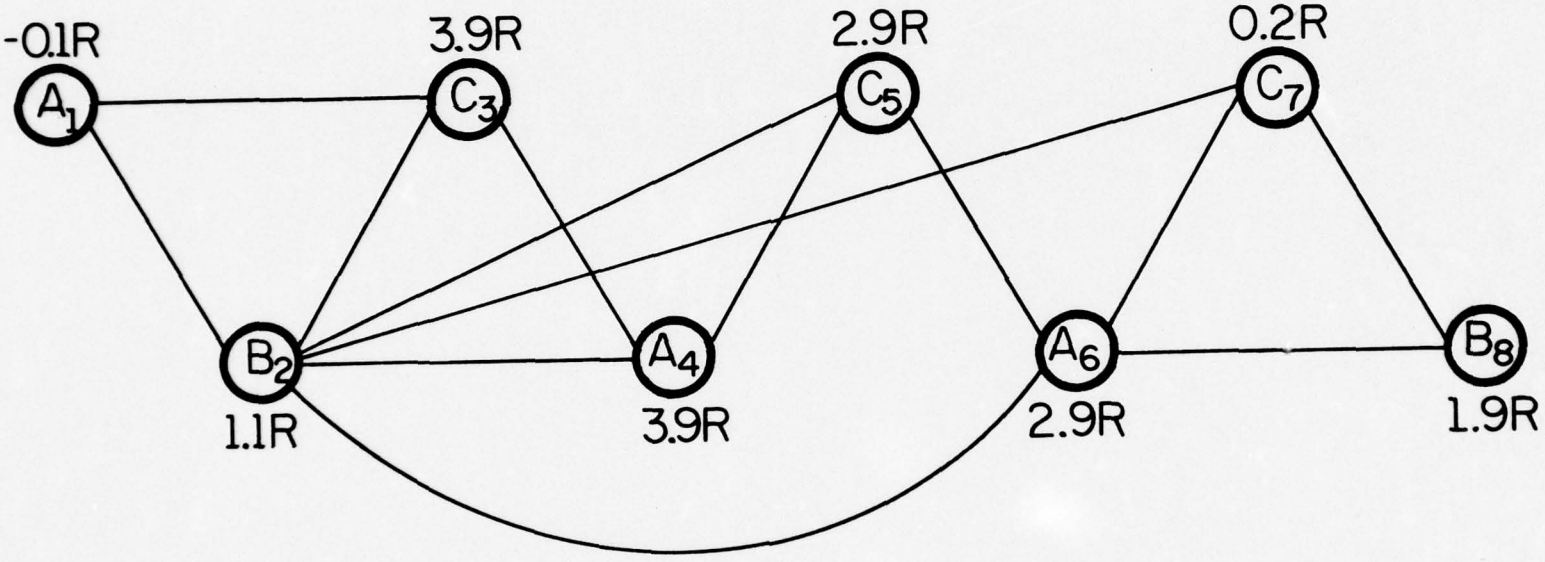
Definition 2.3.3 A subgraph U of $G = [N_G, E_G]$ is the graph $U = [N_U, E_U]$ where $N_U \subseteq N_G$ and $e \in E_U$ if $e \in E_G$ and both nodes in the unordered pair defining e are in N_U.

Definition 2.3.4 The weight of a graph G, WT(G), is

$$\sum_{\forall i \in N_G} NW(i) \quad - \quad \sum_{\forall e \in E_G} EW(i) \ .$$

The weight of the graph in Figure 3 is 16.7R - 13R = 3.7R. The weight of the subgraph, U, with $N_U = \{A_1, A_4, C_5, A_6\}$ is 9.6R - 2R = 7.6R

Let G_{OUT} be the subgraph formed from nodes corresponding to intervals in an OUT set. We assert the following:



FP-5332

Figure 3. A G-graph.

Theorem 2.3.1 $WT(G_{OUT}) = C_{\Delta}(OUT)$.

Proof: The weight of the G_{OUT} subgraph is:

$$\sum_{\forall i \in N_{G_{OUT}}} NW(i) \quad - \quad |E_{G_{OUT}}| * R \ .$$

By construction we have that

$$\sum_{\forall i \in N_{G_{OUT}}} NW(i) \quad = \quad \sum_{\forall i \in OUT} \Delta_i \ .$$

Consider a node $y \in N_{G_{OUT}}$. The edges in G_{OUT} incident on y are connected to nodes in either the set $P_y \cap OUT$ or in a set of the form $P_j \cap OUT$ where $y \in P_j$ and $j \in N_{G_{OUT}}$. Every edge in $E_{G_{OUT}}$ is an unordered pair of the form (a,b) where $a, b \in N_{G_{OUT}}$ and $a \in P_b \cap OUT$. There is exactly one unordered pair for each edge in $E_{G_{OUT}}$. Thus

$$|E_{G_{OUT}}| \quad = \quad \sum_{i \in N_{G_{OUT}}} |P_i \cap OUT| \quad = \quad \sum_{i \in OUT} |P_i \cap OUT| \ .$$

Thus $WT(G_{OUT}) = \sum_{i \in OUT} \Delta_i \quad - \quad \sum_{i \in OUT} |P_i \cap OUT| * R \quad = \quad C_{\Delta}(OUT)$. □

The G-graph is formed from occupied intervals in the starting allocation. In order to find the optimal allocation, it is necessary to determine which of the occupied intervals from the starting allocation are vacated in the optimal allocation. In the graph domain, this operation is equivalent to finding the subgraph of nodes corresponding to intervals that

are vacated in the optimal allocation. Call this subgraph $G_{OUT}*$. The $G_{OUT}*$ subgraph has the following property:

Theorem 2.3.2 The $G_{OUT}*$ subgraph is a minimum weight subgraph of G.

Proof: Let C* be the space-time cost of the optimal allocation. Then

$$C* = C_s + WT(G_{OUT}*) .$$

Form a subgraph A which is different from $G_{OUT}*$. Then the allocation resulting from vacating the intervals in the N_A set has weight:

$C_s + WT(A) \geq C* = C_s + WT(G_{OUT}*)$. Thus $WT(A) \geq WT(G_{OUT}*)$. □

A minimum weight subgraph of G is not unique in general. A minimum subgraph is not unique if there is a zero weight subgraph (ZWS) either wholly within the minimum subgraph or wholly outside of it. Such a ZWS can be placed either wholly within the minimum subgraph or wholly outside of it without affecting the space-time cost of the optimal allocation. Thus, we can reduce space in exchange for increased run time by vacating the intervals in a ZWS, or we can reduce run time in exchange for increased space usage by occupying the intervals in a ZWS. This implies that by vacating the intervals in a negative weight subgraph of G, the space used is decreased by a factor greater than the accompanying increase in run time.

2.4. The Network Flow Graph

The problem is now reduced to finding a minimum weight subgraph. We can solve this problem in a systematic method by transforming the graph

developed in Section 2.3, after reductions, into a flow graph. A similar transformation is found in [15] for a processor scheduling problem. In this section we construct a commodity flow graph with a source and a sink such that the maximum flow from source to sink in this graph has a value equal to the minimum weight subgraph plus a known additive constant. Since there are efficient ways of finding maximum flows, and in so doing, the flow identifies which nodes are in the OUT* set, we can solve the problem of minimizing the page seconds of occupancy that we originally set out to solve.

In constructing a graph for which the maximal flow will give us the desired answer, we remind the reader that the maximal flow is equal to the weight of the minimal cut set of edges that separates the source from the sink. It is this cut set that distinguishes the OUT* set from the remaining nodes by separating the graph into two disjoint subgraphs, one for the OUT* set and the other for nodes not in the OUT* set. The graph described in the previous section has some of the properties that we desire in commodity flow graphs in that some but not all cuts have weights that are in agreement with the cost of the corresponding assignment of intervals to the OUT* set. In order to create a graph whose cut sets are in one-to-one correspondence with the costs of intervals in the OUT* set, we construct a graph derived from a G-graph according to the construction procedure given below. Following the statement of the construction procedure we give an intuitive description of why this construction is appropriate, then we prove that a maximal flow and corresponding minimal cut in the graph produced below does indeed indicate the OUT* set producing minimal page seconds of residency. Consider the following transformation:

Definition 2.4.1 The FG Transform, FG(G), transforms a G-graph, $[N_G, E_G]$, into a flow graph FG(G), by the following procedure:

1. Add two zero weight nodes, a source node, S, and a terminal node, T.
2. Add a zero weight node, called a link node, for each edge in E_G. Cut each edge in E_G into two edges each of weight R. Connect the cut ends to its corresponding link node. These edges are then called link edges.
3. Connect a directed edge with weight R from the source node to each link node having direction from the source to the link node.
4. Connect a directed edge from each node in N_G to the T node having direction from the node to T. Set the weight of the edge equal to the weight of the node in N_G.

Intuitively speaking, Step 1 creates the source and terminal nodes needed to initiate and terminate a flow. The unusual aspect of the construction pertains to the insertion of the link nodes. That these are required are indicated in Figure 4(a), (b), and (c). In Figure 4, we see three possible cuts that respectively assign nodes x_1 and y_2 to T, split x_1 and y_2, and assign x_1 and y_2 to S. Later we shall associate those nodes assigned to S with the intervals in the OUT* set. The value of the cut set in 4(a) is zero, since neither x_1 nor y_2 contributes to change in the value of the assignment, since neither is moved to the OUT* set. In Figure 4(b), we wish to increase the value of the assignment by Δ_{x_1}, since x_1 is moved to the OUT* set. In Figure 4(c), we wish to change the value of the assignment by $\Delta_{x_1} + \Delta_{y_2}$ since both are assigned to the OUT* set. However, the actual increase should be $\Delta_{x_1} + \Delta_{y_2} - R$ if the intervals x_1 and y_2 overlap. We observed earlier that the page fault penalty for one of x_1 or y_2 will be reduced by R page-seconds since the other interval is vacated when the fault occurs.

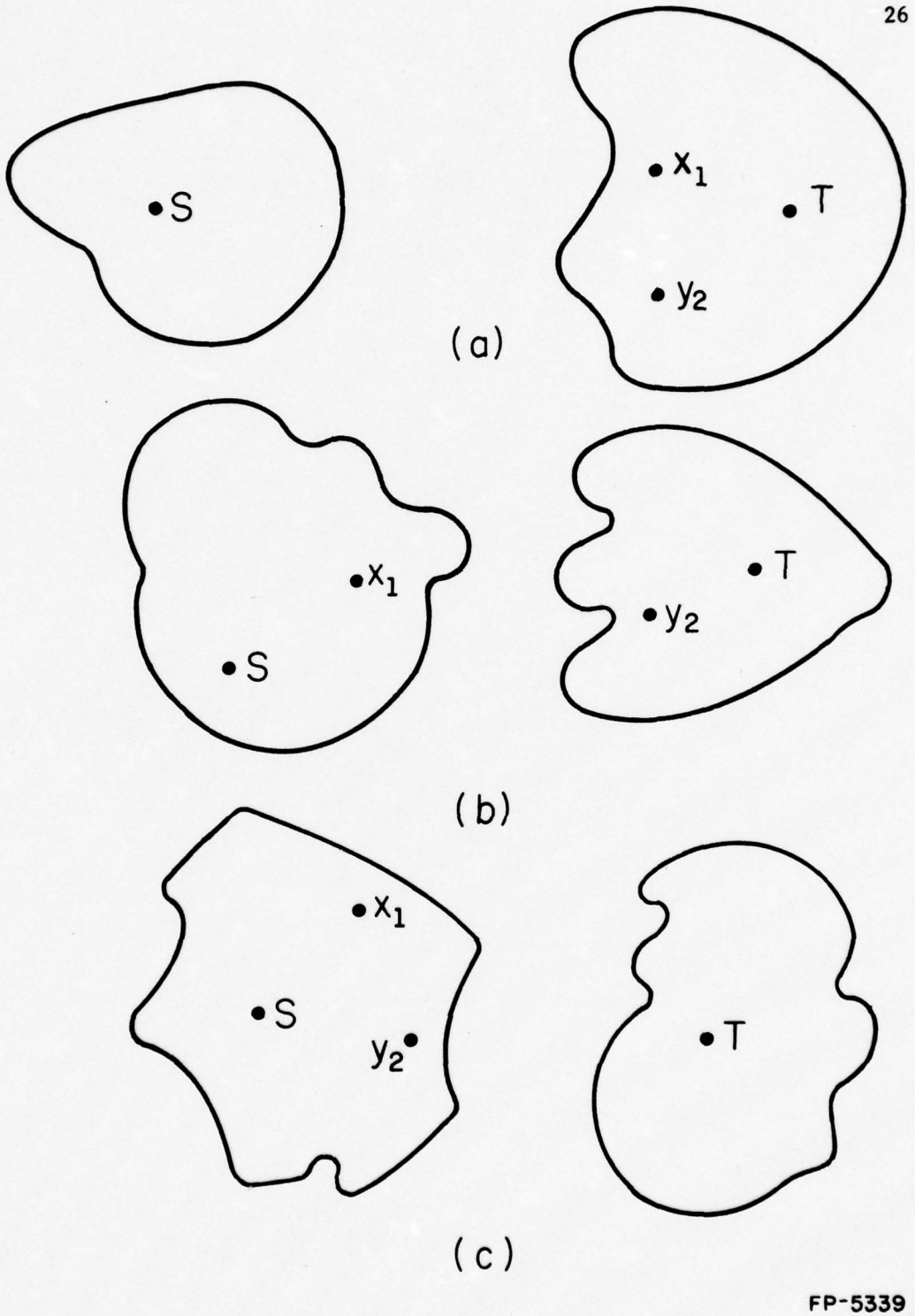


Figure 4. Three possible assignments of intervals

To obtain the value of Δ_{x_1} when x_1 is assigned to node S, we can merely tie node x_1 to T with a branch of weight Δ_{x_1}, which is cut if x_1 is assigned to S and not cut if x_1 is assigned to T. This is done in Step 4 of the graph construction. However this construction by itself does not deduct R from the weight of the cut set when two overlapping intervals are both assigned to S as in Figure 4(c). To see how the construction takes care of this situation, consider Figure 5(a), (b), (c), (d), and (e). We show that we achieve the net effect of reduction of R in case (e) of Figure 5, by instead increasing cases (c) and (d) by R. Observe in Figure 5(c) and (d) that a link edge of weight R is cut when one or both nodes of an overlapping pair of intervals are assigned to T. However, when neither node is assigned to T, no link edge is cut, and we obtain the necessary relative reduction by an amount R. In all three cases, $WT(G_{OUT})$ is R less than the weight of the cut set.

A minimum weight subgraph of G can be found by finding a minimum cut set of the FG(G) graph (to be shown in Corollary 2.4.1). To proceed formally:

Definition 2.4.2 A cut set is a set of edges which, if removed, blocks all paths from the S node to the T node in the FG(G) graph. Furthermore, the cut set partitions the nodes of the graph into two sets: X and $\overline{X}$, where $S \in X$, $T \in \overline{X}$, and all nodes in X are reachable from the S node while all nodes in $\overline{X}$ are not reachable from the S node. A cut set is denoted by $(X,\overline{X})$. The weight of a cut set is the sum of the weights of its edges. A minimum cut set is a cut set with minimum weight.

Definition 2.4.3 For a given cut set, let $\underline{L} = [N_L, E_L]$ be the subgraph of G with $N_L = X \cap N_G$, where X is determined as in Definition 2.4.2.

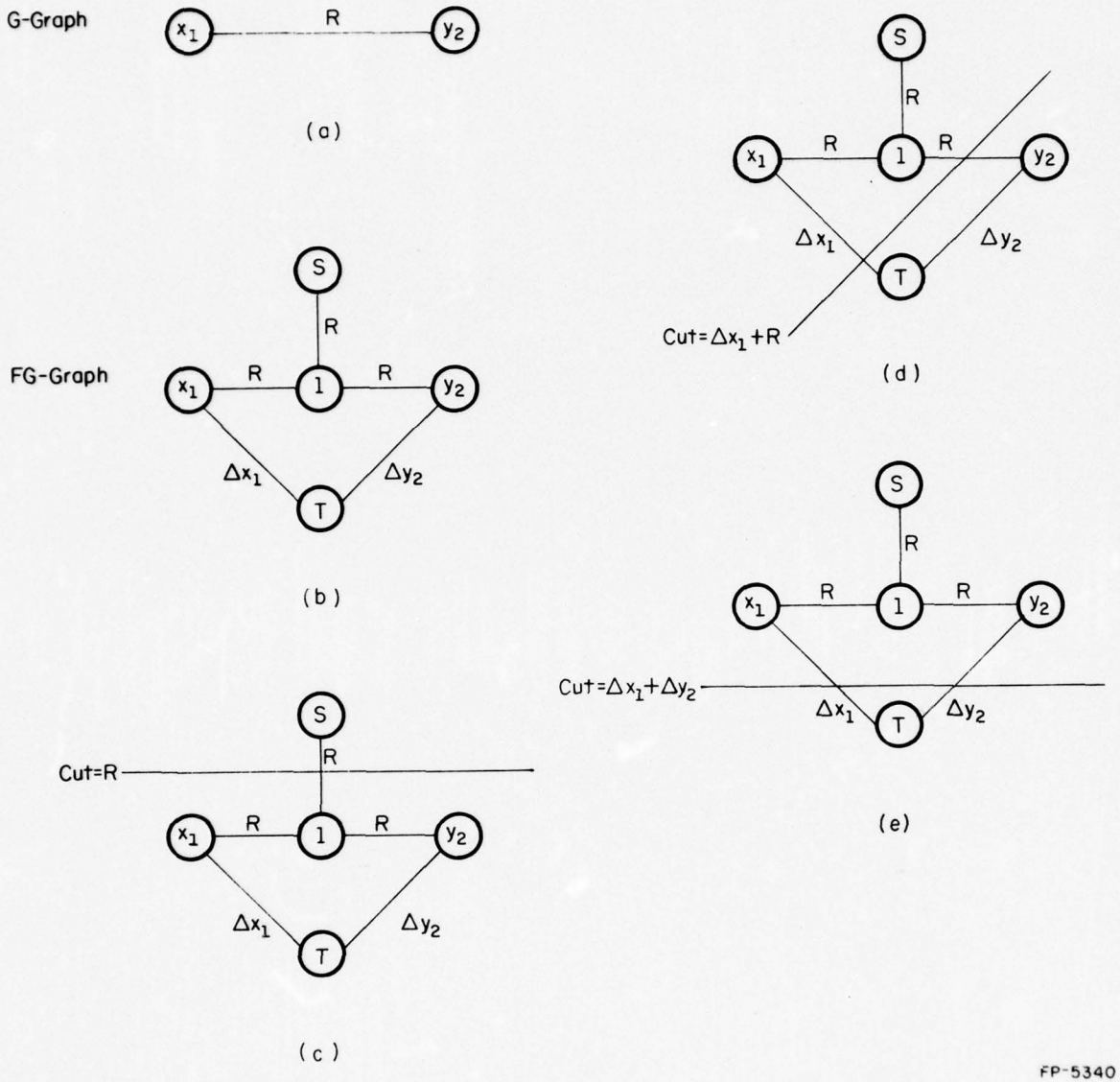


Figure 5. Graphs and cut sets for three assignments of intervals.

FP-5340

Consider the following relationship between a minimum cut set and its corresponding subgraph, L^*.

Theorem 2.4.1 The weight of a minimum cut set of FG(G) is:

$$WT(L^*) + |E_G|*R ,$$

where L^* is the subgraph with $N_{L^*} = X \cap N_G$ and $(X,\overline{X})$ is the partition produced by the minimum cut set.

Proof: By definition, we have $S \in X$, $T \in \overline{X}$. Thus the cut set $(X,\overline{X})$ is not empty. First we deduce link node placement relative to a minimum cut set. Then we can compute the weight of a minimum cut set.

Consider a link node connected to two nodes in L^* through its edges. Call the set of all such nodes the Q set. To form a minimum cut set, each node in Q must be in X. If $y \in Q$ is in X, then none of the three edges incident on y are in $(X,\overline{X})$. If y were in $\overline{X}$, this would add a weight of 3R to the cut set which could not then be a minimum weight cut set. In Figure 5(e), the nodes in L^* are x_1 and y_2 and link node 1 is in the Q set.

Let $\overline{L}^*$ be the subgraph of G formed with all nodes in $\overline{X} \cap N_G$. Let I be the set of link nodes which have one link edge connected to a node in L^*, and the other link edge connected to a node in $\overline{L}^*$. To form a minimum cut set, nodes in I are also in X. This follows because if $y \in I$ is in X, one of its edges is in $(X,\overline{X})$. However, if $y \in \overline{X}$, then two of its edges are in $(X,\overline{X})$. Therefore, the nodes of I are in X. In Figure 5(d), x_1 is in L^* and y_2 is in $\overline{L}^*$. Link node 1 is in the I set.

Let C be the set of link nodes that have both link edges connected to nodes in $\overline{L}^*$. If $y \in C$ is in $\overline{X}$, one of the edges incident on Y is in $(X,\overline{X})$.

However, if $y \in X$, two edges incident on Y would be in $(X,\bar{X})$. Therefore nodes in C are in $\bar{X}$. In Figure 5(c), nodes x_1 and y_2 are in $\bar{L}^*$. Link node 1 is in the C set.

None of the edges in FG between nodes of $\bar{L}^*$ and T are in $(X,\bar{X})$. Also none of the edges incident on any node in Q are in $(X,\bar{X})$. For each node in L^*, the edge from the node in L^* to the T node is in $(X,\bar{X})$. For each node in C or I, one edge of weight R is in $(X,\bar{X})$. Thus, the weight of $(X,\bar{X})$ is:

$$\sum_{n \in N_{L^*}} NW(n) + (|C| + |I|)*R$$

$$= \left(\sum_{n \in N_{L^*}} NW(n) - |Q|*R\right) + (|Q| + |C| + |I|)*R$$

$$= WT(L^*) + |E_G|*R \; . \qquad \square$$

Two terms form the weight of the minimum cut set: a constant term and the weight of the L^* subgraph. If we place link nodes as in Theorem 2.4.1, the weight of any minimum cut set of FG will be the sum of the weight of its corresponding L^* subgraph of G and $|E_G|*R$. Thus, if we place the link nodes as in Theorem 2.4.1, the L^* subgraph must be a minimum weight subgraph:

Corollary 2.4.1 The L^* subgraph is a minimum weight subgraph of G, G_{OUT^*}.

Proof: Consider any subgraph $L' = [N_{L'}, E_{L'}]$ of G. Form a cut set of the FG(G) graph such that $N_{L'} = X' \cap N_G$. Such a cut set can always be formed. Consider placing the link nodes into the X' and $\bar{X}'$ sets as was done in Theorem 2.4.1. This procedure produces the minimum weight cut set consistent

with L'. Then using the same reasoning as in the Theorem we have that the weight of this cut set is:

$$WT(L') + |E_G|*R \geq WT(L^*) + |E_G|*R \ .$$

This implies that:

$$WT(L') \geq WT(L^*) \ . \qquad \square$$

In Figure 6 we construct the FG graph for the G-graph of Figure 3. In this illustration, all edges have weight R except the dashed edges. The weight of a dashed edge shown is adjacent to the edge. We did not attempt to label the graph with the maximum flow since it is already cluttered. However, we have labeled a maximum flow in Figure 7 using an alternate way of drawing the graph. In the graph of Figure 7 we have distributed the source among the edges and the terminal among the nodes. In going from Figure 6 to Figure 7 we have broken a "supersource" and "supersink" (using the terminology of [7]) into distributed sources and terminals. Each edge has an edge source with a positive flow output capacity of R. Each node has a sink of capacity equal to its node weight.

Negative capacity sinks can be accounted for in two ways. One way is to remove the associated node. This method is discussed in Chapter 3. The other way is to assume the existence of negative flow along with positive flow. Every node with negative weight is in every minimum subgraph. Thus the weight of the negative node is added to the weight of the minimum subgraph. In Theorem 2.4.1, the weight of a node is accounted for with an edge from the node to the terminal. But if a node is in the minimum

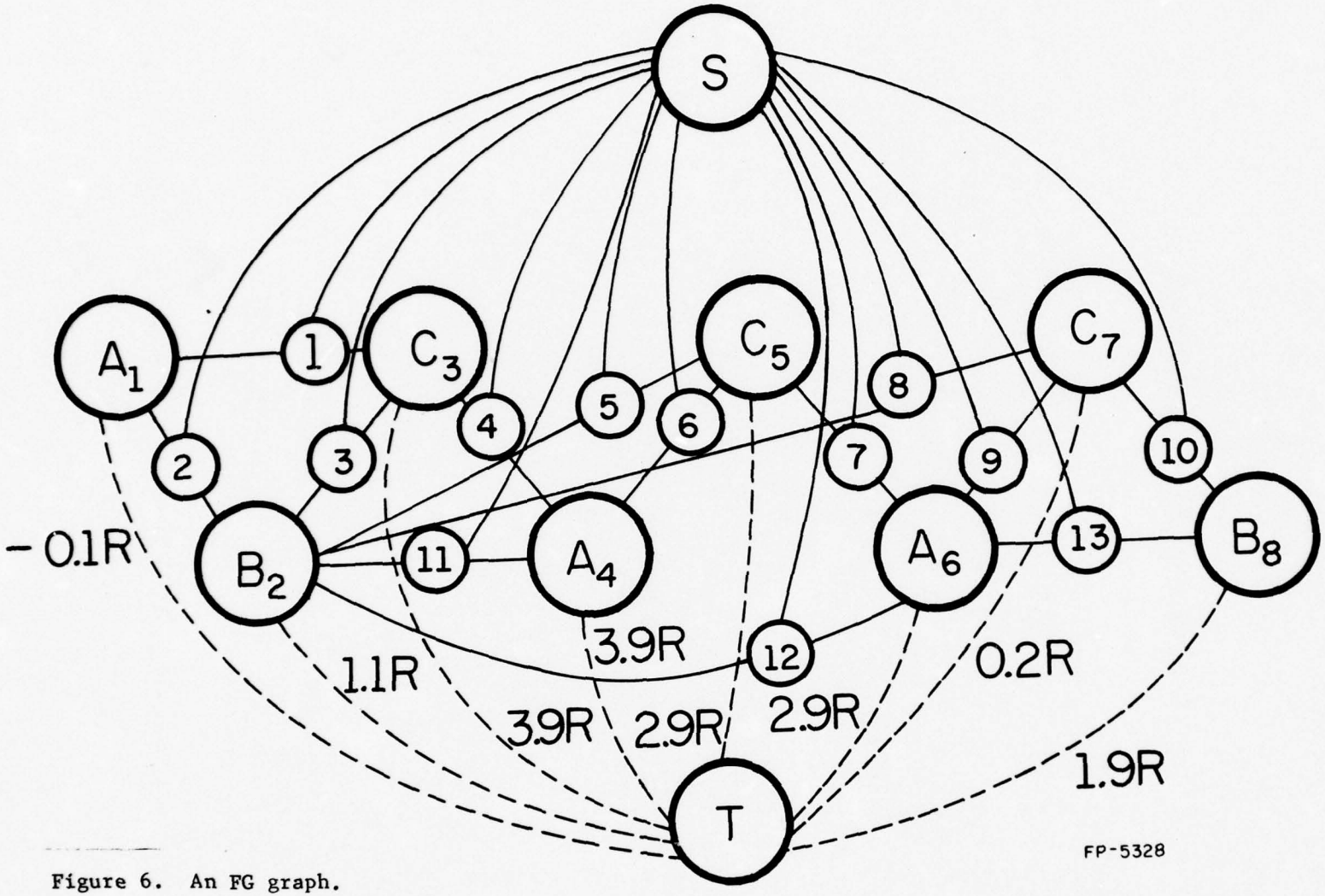


Figure 6. An FG graph.

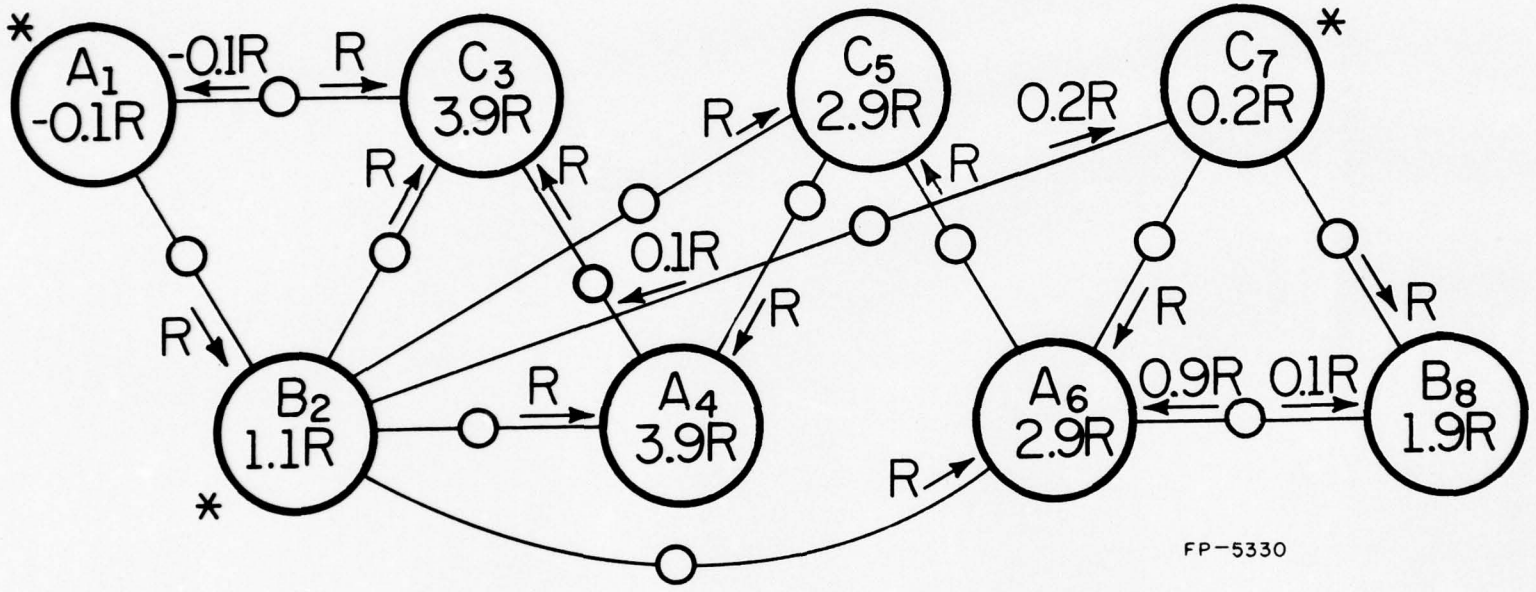


Figure 7. A distributed source and sink flow graph.

subgraph, its terminal edge is in the minimum cut set. Consider all the flow through the edges in a minimum cut set. The value of this flow is equal to the maximum flow. Thus, the flow through a terminal edge of a negative node adds its capacity of negative flow to the maximum flow. Note, this is equivalent to adding the weight of the negative weight node to the minimum subgraph weight. The terminal edge of a negative weight node has zero positive flow capacity. Similarly, positive weight terminal edges have zero negative flow capacity. Link edges may carry any value of flow from $-\infty$ to R in either direction. Edge sources may output any flow from $-\infty$ to R.

In Figure 7 the value of positive maximum flow is 12.3R. The value of negative maximum flow is -.1R. Thus the maximum flow has a value of 12.2R. The nodes with a * shown adjacent to them are in the minimum subgraph. A node is in the minimum subgraph if its weight is negative or it is reachable with flow from some unsaturated edge source along an unsaturated path. Thus node A_1 is in the minimum subgraph because its node weight is negative. Nodes B_2 and C_7 are in the minimum subgraph because the source on the edge between them has output flow capacity left. Also, all other outbound edges from B_2 and C_7 are saturated. Thus, there are no more unsaturated paths left, i.e., no flow augmenting paths exist.

We now know the minimum subgraph for the graph of Figure 6. Thus, consider the Q, I, and C sets of Theorem 2.4.1 for the Figure 6 graph. The Q set consists of the following link nodes: {2,8}. The I set contains link nodes {1,3,5,9,10,11,12}. The C set contains link nodes {4,6,7,13}. The L^* set is $\{A_1,B_2,C_7\}$. The $\overline{L}^*$ set is $\{C_3,A_4,C_5,A_6,B_8\}$. The weight of the minimum subgraph is 12.2R - 13R = -.8R = -.1R + 1.1R + .2R - 2R. Another flow graph

example is found in Figure 10 in Section 3.2. There, the minimum subgraph of the Figure 3 graph is found using additional procedures.

Now the basic steps in the DMIN algorithm can be stated. First, the Δ's and P's are computed from the memory reference trace and the G-graph constructed. The FG(G) graph is formed directly from the G-graph. The minimum subgraph and its weight are then found by determining the maximum flow in the FG(G) graph. From the max flow, min cut theorem [8], the value of the maximum flow is equal to the weight of the minimum cut set. By applying one of the maximum flow algorithms [9-12], both the minimum subgraph and its weight can be found. Let MXFLO be the value of the maximum flow obtained for some FG(G) graph. The space-time cost of the optimal allocation is then C_s + MXFLO - $|E_G|$*R. The optimal dynamic allocation is found by vacating those intervals represented by nodes in the minimum subgraph found.

2.5. The DMIN Allocations as a Function of Reactivation Time

Intuitively, it seems that as reactivation time increases, the number of page faults scheduled by DMIN for a particular program would tend to decrease. With increasing R, the space-time cost increases for processing a page fault. Thus, it seems that circumstances under which a page fault saves space-time cost become more rare as R increases. Consider the following definitions.

Definition 2.5.1 Let G(R) be the graph as constructed using Definition 2.3.2, with reactivation time R.

Definition 2.5.2 Let $L^*(R)$ be any minimum subgraph of $G(R)$.

Definition 2.5.3 Let $\gamma(R_2,R_1)$ be the subgraph of $G(R_2)$ formed by taking the nodes $N_{G(R_2)} - N_{L^*(R_1)}$, where for each edge of $G(R_2)$ in the set $(N_{L^*(R_1)}, N_{G(R_2)} - N_{L^*(R_1)})$, the node in $N_{G(R_2)} - N_{L^*(R_1)}$ connected to such an edge has its node weight reduced by R_2; i.e., $\gamma(R_2,R_1)$ is the remaining subgraph of $G(R_2)$ after the nodes of $L^*(R_1)$ are assigned to be vacated.

Definition 2.5.4 Let $\alpha_i(R_2,R_1)$ be $(2 + |P_i| - |F_i|)$ where the P_i set and the F_i set are determined by occupying the intervals whose corresponding nodes in $G(R_2)$ are in $\gamma(R_2,R_1)$ and vacating intervals whose corresponding nodes in $G(R_2)$ are in $L^*(R_1)$.

With these definitions, we can now prove the following lemma:

Lemma 2.5.1 If $R_1 < R_2$, every subgraph of $\gamma(R_2,R_1)$ has positive weight.

Proof: Consider the weight of any subgraph, H, in $\gamma(R_2,R_1)$. This weight can be expressed as:

$$\sum_{i \in N_H} \alpha_i(R_2,R_1)*R_2 \quad - \quad \sum_{i \in N_H} \tau_i \quad - \quad |E_H|*R_2 \ .$$

Consider $\alpha_i(R_1,R_1)$. It is identical to $\alpha_i(R_2,R_1)$. This statement follows from the fact that $G(R_1)$ and $G(R_2)$ have the same nodes and edges. $G(R_1)$ and $G(R_2)$ differ only in node weights and edge weights. Therefore, $\gamma(R_1,R_1)$ has the same nodes as $\gamma(R_2,R_1)$. Thus, the P_i sets and F_i sets forming $\alpha_i(R_1,R_1)$ are the same sets forming $\alpha_i(R_2,R_1)$. Consider $\gamma(R_1,R_1)$. The

nodes in this graph are the nodes in $G(R_1)$ which are not in the minimum subgraph used to define $\gamma(R_1,R_1)$. Thus, every subgraph of $\gamma(R_1,R_1)$ must have nonnegative weight, since a subgraph with negative weight or a non-empty subgraph of this subgraph must be in all minimum subgraphs of $G(R_1)$.

From the above discussion, we can now express the weight of H as:

$$[(\sum_{i \in N_H} \alpha_i(R_1,R_1)) - |E_H|]*R_2 - \sum_{i \in N_H} \tau_i .$$

But since $\gamma(R_1,R_1)$ has no subgraph with negative weight we know that:

$$[(\sum_{i \in N_H} \alpha_i(R_1,R_1)) - |E_H|]*R_1 - \sum_{i \in N_H} \tau_i \geq 0, \sum_{i \in N_H} \tau_i > 0, \text{ and } (\sum_{i \in N_H} \alpha_i(R_1,R_1)) - |E_H|] > 0.$$

But since $R_2 > R_1$, we have that:

$$[(\sum_{i \in N_H} \alpha_i(R_1,R_1)) - |E_H|]*R_2 > [(\sum_{i \in N_H} \alpha_i(R_1,R_1)) - |E_H|]*R_1 .$$

Therefore we have that:

$$[(\sum_{i \in N_H} \alpha_i(R_1,R_1)) - |E_H|]*R_2 - \sum_{i \in N_H} \tau_i > 0 .$$

Thus, every subgraph of $\gamma(R_2,R_1)$ has positive weight. □

Theorem 2.5.1 If $R_1 < R_2$, then every $L^*(R_2)$ is a subgraph of every $L^*(R_1)$.

Proof: From Lemma 2.5.1, all subgraphs of $\gamma(R_2,R_1)$ have positive weight. Even if only a subgraph of $L^*(R_1)$ were removed, the weights of nodes in

$N_{G(R_2)} - N_{L^*(R_1)}$ could not be less than their weight in $\gamma(R_2,R_1)$. Thus all subgraphs of $\gamma(R_2,R_1)$ would have positive weight even if $\gamma(R_2,R_1)$ were adjusted for only removing part (or none) of $L^*(R_1)$. Therefore no nodes of $\gamma(R_2,R_1)$ may be in any $L^*(R_2)$. Thus $L^*(R_2) \subseteq L^*(R_1)$. □

The usefulness of this result is that once an optimal allocation, $L^*(R_1)$, has been computed for reactivation time R_1, the optimal allocation for $R_2 > R_1$ can be determined as follows. First recompute the node and edge weights of $L^*(R_1)$ for R_2. Then find a minimum subgraph of the adjusted $L^*(R_1)$. Thus, the entire G-graph need not be reconsidered.

From a computational standpoint, the set of intervals vacated by a DMIN allocation may be placed in a stack whose depth is decreased as the reactivation time is increased. This stack structure is valid since as R increases, some intervals will change from the vacated state to the occupied state in the optimal allocation for the larger R, but no previously occupied interval will become vacated. Thus, DMIN is a "stack algorithm" in the sense of Mattson, et al., [13].

In Table 1, we illustrate the effect of increasing the reactivation time for our example. For the first row we show the nodes in the minimum subgraph for a reactivation time used in the examples, i.e., R. The two negative weight subgraphs contain $\{A_1\}$ and $\{A_1,B_2,C_7\}$. As R' is increased, when R' reaches 1.05R, $\{A_1\}$ has weight 0 and the only negative weight subgraph is $\{A_1,B_2,C_7\}$. For reactivation time, R', between R and 1.16R, the nodes in the minimum subgraph are the same, for this example. At R' = 1.16R, the weight of the minimum subgraph becomes zero. Thus there are two minimum subgraphs: the null graph and the minimum subgraph from the

Table 1. The Minimum Subgraphs of G as a Function of Reactivation Time

REACTIVATION TIME R'	MINIMUM SUBGRAPH $N_L^*(R')$	WEIGHT OF MINIMUM SUBGRAPH $WT(N_L^*(R'))$
R	$\{A_1, B_2, C_7\}$	-.8R
1.16R	$\{A_1, B_2, C_7\}$	0
$\beta > 1.16R$	ϕ	0

example. In the table we choose to use the minimum subgraph from the example. For any reactivation time exceeding 1.16R, the minimum subgraph is the null graph.

In Figure 8 we illustrate the subgraph used to generate Table 1. In Figure 8 the node weights are expressed in terms of R' and R. Note that the τ remain functions of R. For the figure, R' is the variable reactivation time, and R is an arbitrary positive constant. Each of the edges has weight R'.

2.6. Discussion

Given a timed page reference trace for a program's run and assuming a multiprogrammed computer with demand paging, we have presented DMIN, an optimal dynamic allocation algorithm which minimizes the space-time cost of primary memory used during the run. As shown in Chapter 3, the collection of the trace and the formation of the graph are both $\mathcal{O}(N)$ operations, where N is the number of memory references. Computing the maximum flow for graphs like FG(G) will be shown to be $\mathcal{O}(N^3)$ when applying the maximum flow algorithms [9-12]. In Chapter 3 we also present reductions which can be made upon the nonreference intervals.

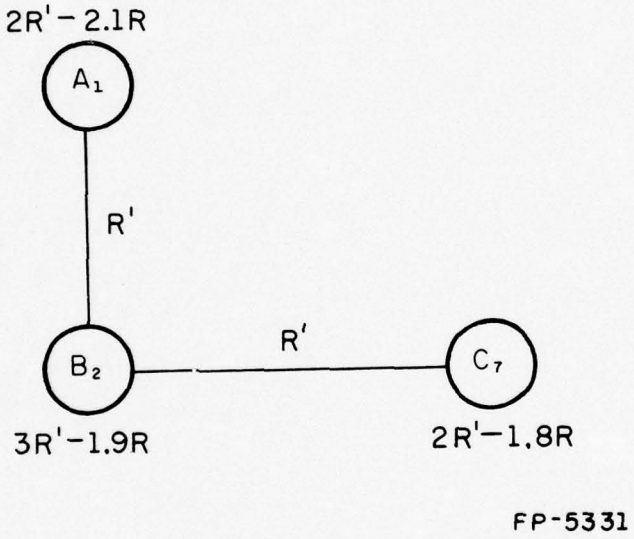


Figure 8. A minimum subgraph as a function of variable reactivation time R'.

CHAPTER 3

IMPLEMENTATION OF DMIN

In order to implement DMIN, three basic operations are performed: gathering a page reference trace, forming a flow graph, and finding the maximum flow in the graph. In the worst case, the first two operations require $\mathcal{O}(N)$ steps, and the last operation requires $\mathcal{O}(N^3)$ steps where N is the number of page references in the page trace. For a one second execution of a program, N could be as large as several million. Determining the maximum flow in a graph with millions of nodes is impractical if the worst case computational complexity occurs. One method to solve this problem is to reduce the number of nodes that are needed in the flowgraph. This method is developed in Sections 3.1, 3.2, and 3.3. Another method is to separate the flowgraph into disjoint graphs. We develop the disjoint graph method in Section 3.4. Our purpose in this chapter is to formalize these two methods in order to achieve practical implementations of the DMIN algorithm. Also, we shall describe an implementation of DMIN.

3.1 Nonreference Interval Reduction Techniques

In the G-graph, a node can be in one of two states: in the minimum subgraph (vacated), or not in the minimum subgraph (occupied). The state of some nodes can readily be determined. When the state of a node is known, the graph is modified to reflect this knowledge. In terms of intervals, when we know the state of an interval in the optimal allocation, we can assign the interval to that state. Thus, an interval whose node is

in the minimum subgraph also belongs to the OUT* set, the set of intervals whose state is the vacated state in the optimal allocation.

Nodes that have negative weight are definitely in the minimum subgraph. The corresponding intervals of such nodes belong to the OUT* set. If we know that a node has negative weight, we can remove it from the graph. Suppose node y has negative weight. When node y is removed, every node connected to y through a single edge has its node weight reduced by R. To show this, consider a node z and assume an edge, (y,z) exists. In terms of intervals y and z, an edge between their nodes implies that the intervals overlap. Thus either $y \in P_z$ or $z \in P_y$.

If $y \in P_z$, then y should be removed from P_z when it is assigned to be vacated. *If $z \in P_y$, then y should be added to F_z* when y is assigned to be vacated. In either case, by Definition 2.2.4, Δ_z is reduced by R. To justify this change, note that if y is vacated and $y \in P_z$, the page associated with interval y would not be in the memory during the return of the z interval page should we decide to vacate interval z. If y is vacated and $z \in P_y$, then the length of interval z is increased by R when a page fault occurs at the end of interval y.

In general, if a subgraph G' is removed, for each edge $e \in (N_{G'}, N_G - N_{G'})$ (the set of edges between nodes of G' and other nodes of G), the node in $N_G - N_{G'}$ connected to e has its weight reduced by R when G' is removed from the graph. After removing negative weight nodes, other negative weight nodes may be formed as a result of the subgraph removing process. Thus, a procedure to remove negative weight nodes must iterate on a search step followed by a removal step.

We can generalize the detection of single nodes in a minimum subgraph to the detection of subgraphs of a minimum subgraph. A subgraph is a subgraph of a minimum subgraph if two properties are fulfilled. First, no subset of the nodes in the subgraph of a minimum subgraph increases the weight of this subgraph. In other words, we can't remove any nodes in the subgraph without increasing the remaining subgraph's weight. Otherwise this entire subgraph would not be part of a minimum subgraph because some nodes could be removed to decrease the weight of the minimum subgraph. In this case, only part of the subgraph would be in the minimum weight subgraph. Second, the weight of the subgraph of the minimum subgraph is nonpositive. The null graph has zero weight. Thus any minimum subgraph must have nonpositive weight. Applying the first condition to a minimum subgraph implies that no subgraph of the minimum subgraph can increase the minimum subgraph's weight. This condition is fulfilled by a subgraph with nonpositive weight. In terms of intervals, the intervals corresponding to nodes in a nonpositive weight subgraph can be assigned to the vacated state without increasing space-time cost. Consider the following:

Theorem 3.1.1: A subgraph, U, of G is a subgraph of a minimum subgraph of G if two conditions are true:

(i) For all subgraphs g_1 and g_2 such that $N_{g_1} \cup N_{g_2} = N_U$ and $N_{g_1} \cap N_{g_2} = \phi$, we have that: $WT(g_1) - |(g_1,g_2)| * R \leq 0$, where (g_1,g_2) is the set of edges between nodes of g_1 and nodes of g_2.

(ii) $WT(U) \leq 0$.

Proof: If a subgraph U of G satisfies condition (i), no subgraph of U has weight less than the weight of U. We have that:

$$WT(U) = WT(g_1) + WT(g_2) - |(g_1,g_2)|*R.$$

Thus condition (i) implies that

$$WT(U) = WT(g_1) + WT(g_2) - |(g_1,g_2)|*R \leq WT(g_2).$$

Thus if U satisfies condition (i), no subgraph of U can be removed so that the remainder of U has smaller weight.

If neither of conditions (i) and (ii) are satisfied by equality, U must be a subgraph of every minimum subgraph of G. Otherwise, the weight of a "minimum" subgraph not wholly containing U could be reduced by including all of U.

Similarly, if U satisfies (i) and (ii), it must be a subgraph of some minimum subgraph of G. □

If a minimum subgraph, H, is disjoint from a zero weight subgraph, U, which satisfies (i) and (ii) above and has no edges of G between nodes of U and nodes of H, then U may be added to H without change in weight. If a zero weight subgraph, U, which satisfies (i) and (ii) above, has edges of G between its nodes and the other nodes of a minimum weight subgraph, H, then U must be wholly contained within H. Accordingly, we choose to include zero weight subgraphs in constructing our minimum subgraphs.

We propose the following node reduction algorithm for removing subgraphs with i or fewer nodes that are subgraphs of a minimum subgraph.

Algorithm RED(i):

1. Check a subgraph with i nodes for satisfying conditions (i) and (ii) of

Theorem 3.1.1. If an i node subgraph (and all of its subgraphs) fails to satisfy (i) and (ii), go to step 5. If a subgraph with i or fewer nodes satisfies (i) and (ii), go to step 2.

2. Remove the subgraph found in the preceding step.
3. For each remaining node, y, subtract R from the weight of y for each edge connecting y to a node of the removed subgraph.
4. Check all subgraphs with i nodes containing at least one node that has had its weight reduced in step 3. Check each such subgraph as in step 1 for satisfying conditions (i) and (ii) of Theorem 3.1.1. When a subgraph with i or fewer nodes satisfies both these conditions, go to step 2. When all such subgraphs are checked, go to step 5.
5. If unchecked i node subgraphs exist in the remaining graph, go to step 2. Otherwise, RED(i) is complete.

Theorem 3.1.2 The worst case complexity for RED(i) is $\mathcal{O}(DP * N^i)$ where N is the number of nodes in G and DP is the number of distinct pages referenced in the trace which forms G.

Proof: In the worst case, $\binom{N}{i}$ subgraphs are checked in step 1 of RED(i) (for the conditions of Theorem 3.1.1) before a subgraph of a minimum subgraph is found. Checking a subgraph requires no more than $2^i * i^2$ steps. In order to check a subgraph U (for conditions (i) and (ii) of Theorem 3.1.1), we have to consider all subgraphs of U. There are no more than $\sum_{j=1}^{i} \binom{i}{j} < 2^i$ subgraphs of U. Checking a subgraph of U takes at most i^2 steps. There are at most $(i^2-i)/2$ edges for which we assume one step to access and one step to add the edge weights. There are at most i nodes which require (i-1)

additions of node weights and the subtraction of the sum of the edge weights. Thus, it takes no more than $\binom{N}{i}*i^2*2^i$ steps to check the $\binom{N}{i}$ subgraphs.

We assume that removing a subgraph from G in step 2 of RED(i) takes one step for each node in the subgraph and one step for each edge adjacent to a node in the removed subgraph. Thus, removing all nodes takes N steps. There are less than (DP-1)*N edges in G, where DP is the number of distinct pages referenced within the trace. Thus, removing the entire graph takes at most DP*N steps.

When a subgraph is removed, for each edge, e, connecting the removed subgraph to a node, y, in the remainder of the graph, the weight of y is decreased by R in step 3. We assume that it takes one step to access a node weight and one step to subtract R. Since an edge can only be removed once, the adjustment procedure takes at most 2 times the number of edges in G, i.e., no more than 2*(DP-1)*N steps.

A node, y, is in a subgraph to be checked if a node, z, reachable through a path of up to i-1 edges from y is adjacent to a removed subgraph. Consider a node adjusted in step 3. There are no more than $\binom{N-1}{i-1}$ i node subgraphs containing this adjusted node. A node can be adjusted only once for each incident edge. Thus, all the nodes in the graph can initiate at most (DP-1)*N checks in step 4. An i node check takes no more than i^2*2^i steps. Thus there are no more than $N*(DP-1)*\binom{N-1}{i-1}*i^2*2^i$ steps performed in step 4 of RED(i).

The worst case complexity of RED(i) is less than:

$$\binom{N}{i}*i^2*2^i + DP*N + 2*(DP-1)*N + N*(DP-1)*\binom{N-1}{i-1}*i^2*2^i \quad .$$

Consider the first term. We have that:

$$\binom{N}{i}*i^2*2^i < \frac{i^2*2^i}{i!} * N^i \leq 12*N^i,$$

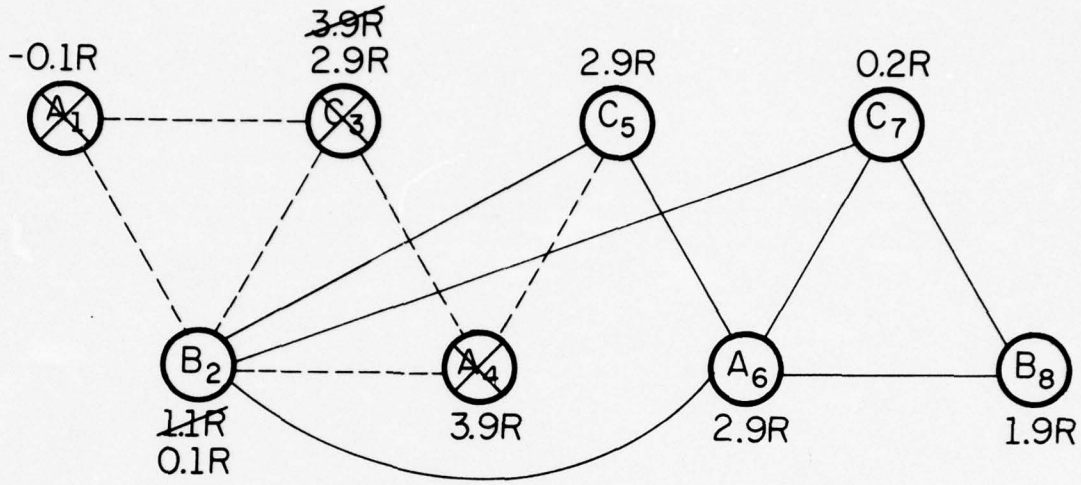
Consider the fourth term. We have that:

$$N*(DP-1)*\binom{N-1}{i-1}*i^2*2^i < (DP-1)* \frac{i^2*2^i}{(i-1)!}* N^i < (DP-1)*43*N^i \quad .$$

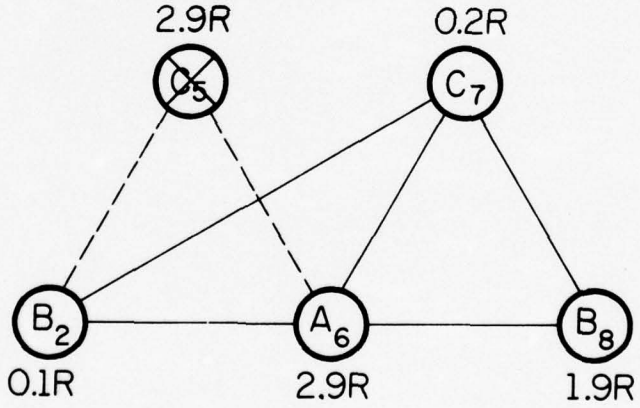
Thus we have that the worst case complexity of RED(i) is $\mathcal{O}(DP*N^i)$. □

Another possible reduction is to determine those nodes that are definitely not in the minimum subgraph. Consider a node, y, whose node weight exceeds the sum of the edge weights incident on y. If y were in a minimum subgraph, its contribution to the subgraph weight would actually increase the weight of the subgraph. The node weight of y minus the weight of all the edges it could possibly add to the minimum subgraph is positive. We use this fact in the following example. Also, this fact motivates the next section.

In Figure 9 we show two stages in the reduction process of the graph from Figure 3 of Chapter 2. RED(1) is applied. Nodes which are removed are crossed out. Edges which are removed are dashed. In Figure 9a, node A_1 is removed since its weight is negative. Thus A_1 is an element of OUT*. In removing A_1, the weight of B_2 is decreased to .1R. The weight of C_3 is similarly reduced to 2.9R. Node C_3 is removed since before A_1 is removed C_3 has weight 3.9R and 3 edges of weight R. Thus, even if the three adjacent nodes of C_3 are in the minimum weight subgraph, the weight of C_3



(a)



(b)

FP-5333

Figure 9. G-graph reduction.

is still positive. Thus in every case, vacating C_3 will increase the space-time cost. Similarly, vacating A_4 will always increase space-time cost. Thus neither A_4 nor C_3 can belong to any minimum subgraph, and therefore intervals C_3 and A_4 are not in any OUT* set. In removing nodes like A_4 and C_3, the weights of adjacent nodes are not changed. No change is made since all node weights were formed assuming A_4 and C_3 are occupied. In removing A_4 and C_3 from the graph, we are definitely assigning them to be occupied.

In Figure 9b, node C_5 is removed since it cannot belong to any minimum subgraph. No further single node reductions are possible.

In Figure 10 we construct the flow graph for the graph remaining after the reduction in Figure 9. In Figure 10, integer-labeled nodes are link nodes. Edges with flow capacity different than R are drawn as dashed lines and labeled with two quantities enclosed in brackets. The quantity on the left is the flow capacity of that edge. The quantity on the right is the actual flow through that edge in a maximum flow assignment. Edges with flow capacity R are drawn as solid lines and labeled by the amount of flow through these edges in the maximum flow assignment. The maximum flow through the graph has a value of 4.3R. The edges in the minimum cut set are cut by a heavy line. The weight of the minimum cut set is 4.3R. The weight of the minimum cut set minus R times the 5 edges in the reduced G-graph of Figure 9b is -.7R. The elements in the minimum subgraph are B_2 and C_7. The weight of the minimum subgraph is:

$$NW(B_2) + NW(C_7) - R = .1R + .2R - R = -.7R \ .$$

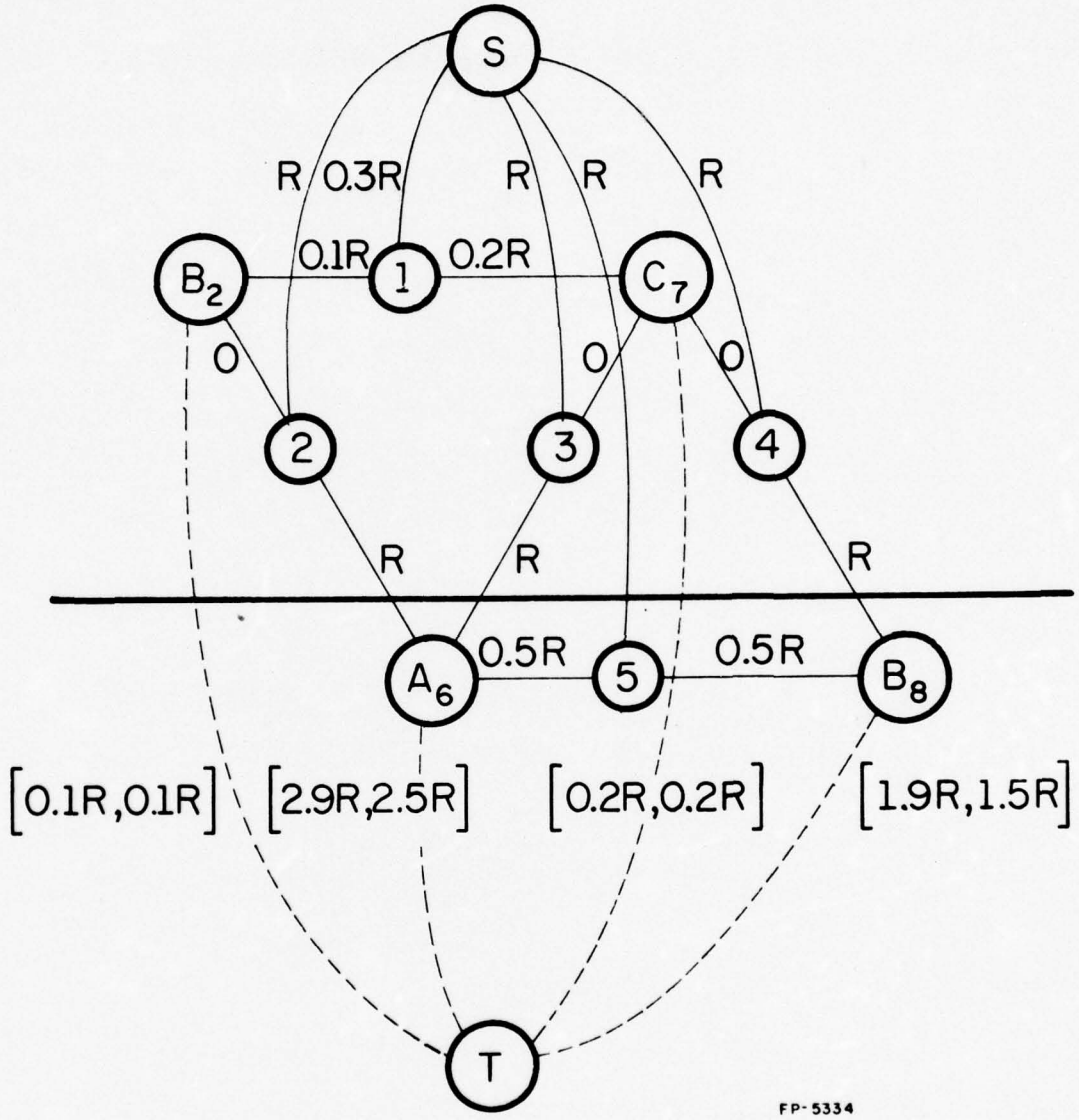


Figure 10. An FG graph.

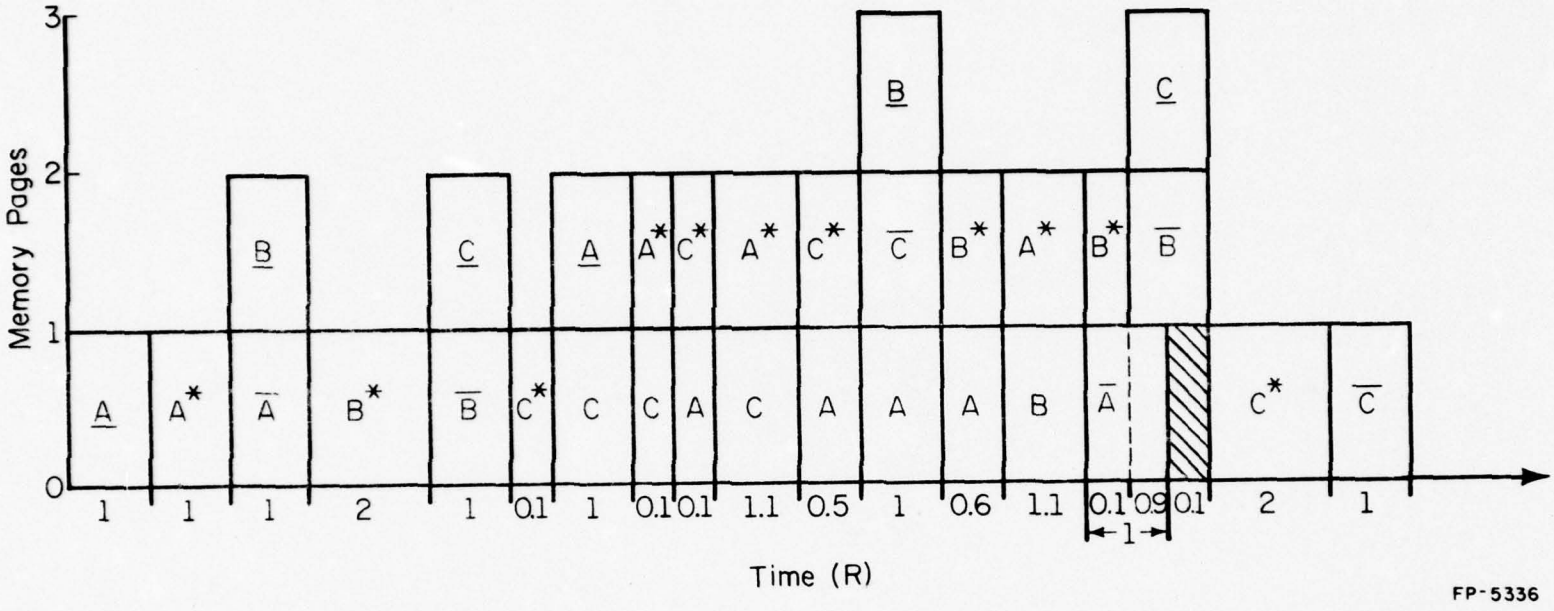
Node A_1 is also in the minimum subgraph. We have already subtracted the weight of the edge between A_1 and B_2 from the weight of B_2. Thus, we need only to add the weight of node A_1. Therefore the weight of the minimum subgraph is $-.7R + -.1R = -.8R$. Note this is the same result as in the example in Chapter 2.

The L* set is $\{B_2, C_7\}$. The $\overline{L}$* set is $\{A_6, B_8\}$. The Q set is $\{1\}$. The I set is $\{2,3,4\}$. The C set is $\{5\}$.

In Figure 11 we plot the allocation profile for the optimal dynamic allocation of our example. In Figure 12 we plot the allocation profile resulting from using Belady's MIN algorithm with a static allocation of 2 pages. In Figure 13 we plot the profile resulting from MIN with a static allocation of 3 pages (similar to the DMIN Starting Allocation except for delayed final deallocation and early reservation). The Starting Allocation has space-time cost $C_s = 27$ R. In these figures, $\underline{A}$ means that page A is being transferred into the primary memory due to a page fault; $\overline{A}$ means that page A is being deallocated. As stated above we set D equal to R. A* means that A is being referenced. A alone means that A is resident and awaiting execution.

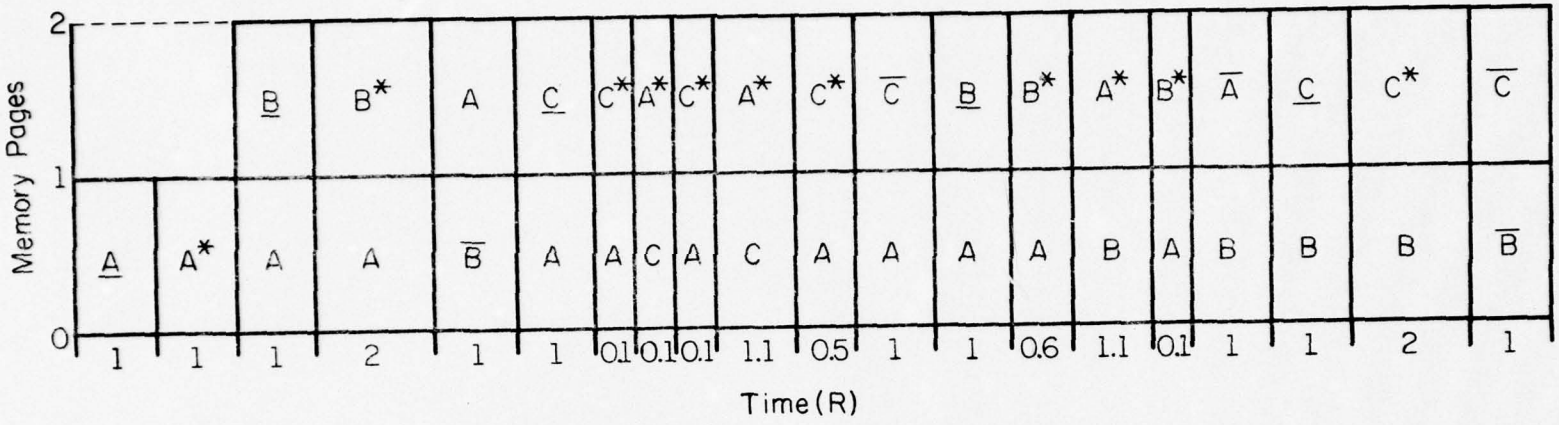
3.2 Development of the Complementary Graph, $\overline{G}$

In the previous example, we saw that certain nodes can be removed from the G graph if they are definitely not in a minimum subgraph of G. In this section we develop a systematic method for determining the nodes which are not in the minimum subgraph. Our method is to parallel the development of Sections 2 and 3 of Chapter 2. The major difference is the starting allocation. In place of the starting allocation of Definition 2.2.1, we use the allocation in which every nonreference interval is vacated.



Completion time: 15.7R
Space-time cost: 26.2R
Page faults: 6

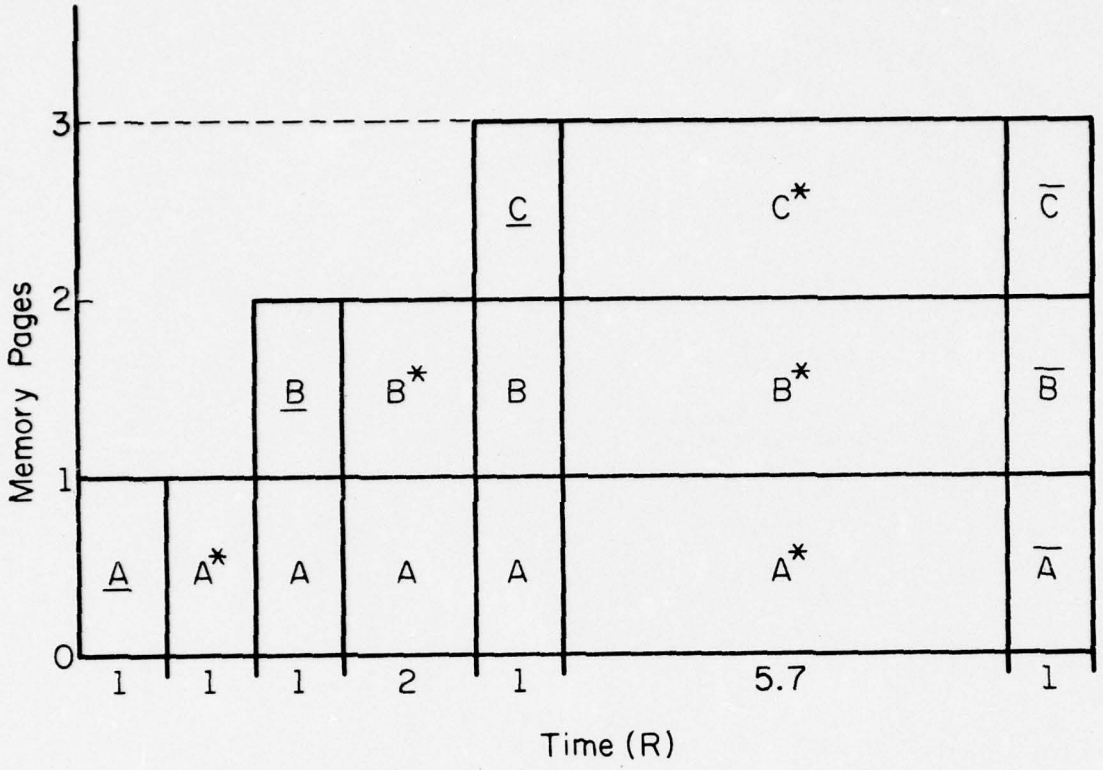
Figure 11. The allocation profile from DMIN.



FP-5338

Completion time: 17.7R
Space•time cost: 35.4R
Page faults: 5

Figure 12. The allocation profile from MIN with two pages allocated.



Completion time: 12.7R
Space-time cost: 38.1R
Page faults: 3

Figure 13. The allocation profile for MIN with three pages allocated.

Definition 3.2.1 The Out Starting Allocation is formed by vacating all nonzero length nonreference intervals. Zero length reference intervals are occupied.

For the i^{th} nonreference interval in the out starting allocation, τ_i is identical to the τ_i from the starting allocation. However, other quantities require new definitions.

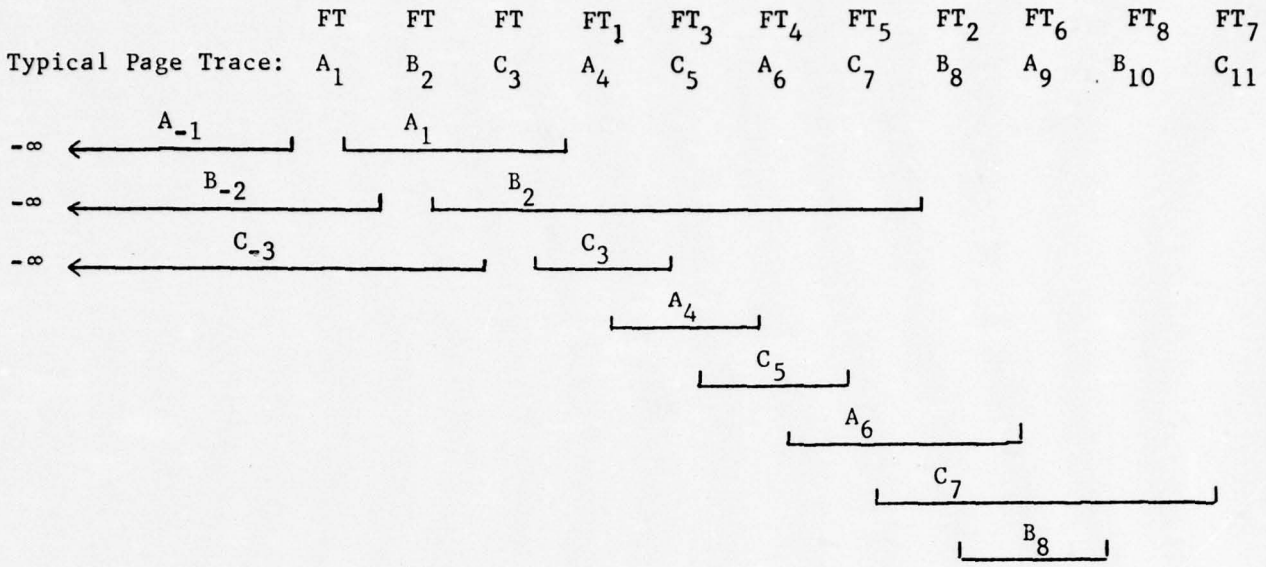
Definition 3.2.2 $\overline{F}_i$ is the set of vacated nonreference intervals which end within the i^{th} nonreference interval for the current allocation which begins with the out starting allocation.

Definition 3.2.3 $\overline{P}_i$ is the set of occupied nonreference intervals overlapping the end of the i^{th} nonreference interval for the current allocation which begins with the out starting allocation.

Definition 3.2.4 The Out Differential Cost for the i^{th} interval is the out starting allocation, $\overline{\Delta}_i$, is

$$(\tau_i + |\overline{F}_i| * R) - (D + R + |\overline{P}_i| * R) .$$

In Figure 14, we apply the difinitions 3.2.1, 3.2.2, 3.2.3, and 3.2.4 to the example of Chapter 2. For the purpose of listing the $\overline{F}_i$ sets, we have named the three initial nonreference intervals A_{-1}, B_{-2}, C_{-3}. These intervals begin at $-\infty$ and are known to be vacated. They are therefore not shown in Figure 14. Intervals B_{-2} and $C_{-3} \in \overline{F}_1$. Also $C_{-3} \in \overline{F}_2$. The other initial $\overline{F}_i$ elements are all finite nonreference intervals which end within



TRACE ELEMENT (Nonreference Interval)	$\overline{F}_i$	$\lvert\overline{F}_i\rvert$	τ_i	$\overline{\Delta}_i$
A_1	$\{B_{-2}, C_{-3}\}$	2	2.1R	2.1R
B_2	$\{C_{-3}, A_1, C_3, A_4, C_5\}$	5	1.9R	4.9R
C_3	$\{A_1\}$	1	.1R	-.9R
A_4	$\{C_3\}$	1	.1R	-.9R
C_5	$\{A_4\}$	1	1.1R	.1R
A_6	$\{C_5, B_2\}$	2	1.1R	1.1R
C_7	$\{B_2, A_6, B_8\}$	3	1.8R	2.8R
B_8	$\{A_6\}$	1	1.1R	.1R
A_9	–	–	∞	$+\infty$
B_{10}	–	–	∞	$+\infty$
C_{11}	–	–	∞	$+\infty$

Figure 14. Properties of nonreference intervals.

the i^{th} interval. The τ_i's are identical to the τ_i's from Figure 2 of Chapter 2. We list no $\bar{P}_i$'s since these are all ϕ for the out starting allocation.

The differential cost, $\bar{\Delta}_i$, is the change in space time cost for changing the i^{th} interval from the vacated state to the occupied state. Let the IN set be a set of nonreference intervals which have their state changed from the vacated state to the occupied state.

Theorem 3.2.1. The change in space time cost for occupying the intervals belonging to the IN set is

$$C_{\bar{\Delta}}(IN) = \sum_{i \in IN} \bar{\Delta}_i - \sum_{i \in IN} | \bar{F}_i \cap IN | *R$$

Proof: Consider an interval $k \in IN$. By changing its state from vacated to occupied we need to replace its cost for vacating the memory with the cost for occupying the memory. In the out starting allocation cost when interval k is vacated, its cost contribution is $D + R + |\bar{P}_i| *R$. For occupying the i^{th} interval, the cost is the real time length of the interval:

$$\tau_i + (| \bar{F}_i | - | \bar{F}_i \cap IN |)*R .$$

Thus for each interval k belonging to IN, the change in cost is:

$$\tau_i + | \bar{F}_i |*R - (D + R + |\bar{P}_i|*R) - | \bar{F}_i \cap IN |*R = \bar{\Delta}_i - | F_i \cap IN |*R .$$

Therefore:

$$C_{\bar{\Delta}}(IN) = \sum_{i \in IN} \bar{\Delta}_i - \sum_{i \in IN} | \bar{F}_i \cap IN |*R .$$

□

As in Chapter 2, we form a graph representation for the non-reference interval model. Consider the following:

Definition 3.3.5 A $\overline{G}$-graph is a graph $\overline{G} = [N_{\overline{G}}, E_{\overline{G}}]$. The nodes of $N_{\overline{G}}$ correspond one-to-one with finite length nonreference intervals. The weight of each node is the $\overline{\Delta}$ for the corresponding nonreference interval. Each node has the same name as its corresponding nonreference interval. An edge is connected between node A_i and node B_j if an only is $A_i \in \overline{F}_j$ or $B_j \in \overline{F}_i$, i.e., one nonreference interval ends within the other one. Every edge has weight R.

In Figure 15 we draw the $\overline{G}$-graph from the nonreference interval properties in Figure 14. The structure of the graph is identical to the G-graph in Figure 3.

Let $\overline{G}_{IN}$ be the subgraph formed from nodes corresponding to intervals in an IN set. We assert the following:

Theorem 3.2.2 $WT(\overline{G}_{IN}) = C_{\overline{\Delta}}(IN)$

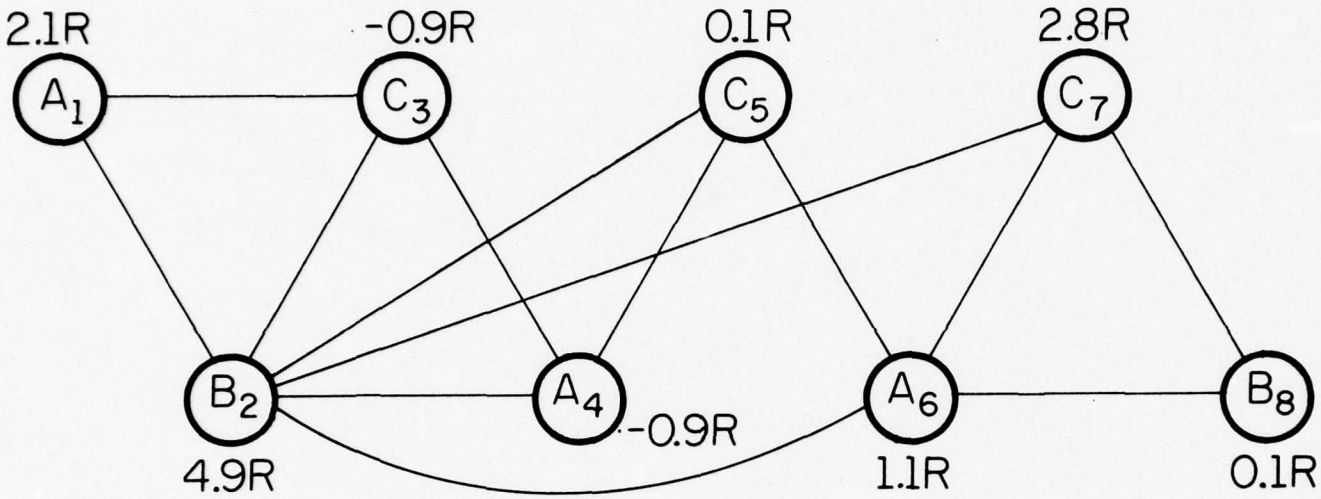
Proof: The weight of the $\overline{G}_{IN}$ subgraph is:

$$\sum_{\forall i \in N_{\overline{G}_{IN}}} NW(i) - |E_{G_{IN}}| * R.$$

By construction we have that

$$\sum_{\forall i \in N_{\overline{G}_{IN}}} NW(i) = \sum_{\forall i \in IN} \overline{\Delta}_i .$$

Consider a node $y \in N_{\overline{G}_{IN}}$. The edges in $\overline{G}_{IN}$ incident on y are connected to nodes either in the set $\overline{F}_y \cap N_{\overline{G}_{IN}}$ or in a set of the form $\overline{F}_j \cap N_{\overline{G}_{IN}}$ where $y \in \overline{F}_j$ and $j \in N_{\overline{G}_{IN}}$. Every edge in $E_{\overline{G}_{IN}}$ is an unordered pair of the form (y,z) where $y, z \in N_{\overline{G}_{IN}}$ and $z \in \overline{F}_y \cap N_{\overline{G}_{IN}}$. There is exactly one unordered



FP-3529

Figure 15. A $\bar{G}$-graph.

pair for each edge in $E_{\bar{G}_{IN}}$. Thus,

$$|E_{\bar{G}_{IN}}| = \sum_{i \in N_{\bar{G}_{IN}}} |\bar{F}_i \cap N_{\bar{G}_{IN}}| = \sum_{i \in IN} |\bar{F}_i \cap IN| .$$

Thus

$$WT(\bar{G}_{IN}) = \sum_{i \in IN} \bar{\Delta}_i - \sum_{i \in IN} |\bar{F}_i \cap IN| * R = C_{\bar{\Delta}}(IN) .$$

□

The $\bar{G}$-graph is formed from vacated intervals in the out starting allocation. In order to find the optimal allocation, it is necessary to determine which of the vacated intervals from the out starting allocation are occupied in the optimal allocation. In the graph domain this operation is equivalent to finding the subgraph of nodes corresponding to intervals which are occupied in the optimal allocation. Call this subgraph $\bar{G}_{IN}*$. The $\bar{G}_{IN}*$ subgraph has the following property:

Theorem 3.2.3 The $\bar{G}_{IN}*$ subgraph is a minimum weight subgraph of $\bar{G}$.

Proof: Let C* be the space-time cost of the optimal allocation, and let $\bar{C}_s$ be the cost of the out starting allocation. Then

$$C* = \bar{C}_s + WT(\bar{G}_{IN*}).$$

Form a subgraph A which is different from $\bar{G}_{IN}*$ then the allocation resulting from occupying intervals in the N_A set has weight:

$$\bar{C}_s + WT(A) \geq C* = \bar{C}_s + WT(\bar{G}_{IN}*).$$

Thus $WT(A) \geq WT(\bar{G}_{IN}*)$. □

A minimum subgraph can be found by using the methods of Chapter 2. First a flow graph $FG(\bar{G})$ is formed. Then Theorem 2.4.1 becomes applicable by replacing G with $\bar{G}$. The method for determining the minimum subgraph $\bar{G}$ is identical to the method for G.

3.3 The G_A-Graph and G_A-Graph Reductions

In the out starting allocation domain, a nonpositive $\bar{\Delta}$ corresponds to a node which is occupied in the optimal allocation. Occupying a non-reference interval whose $\bar{\Delta}$ is negative saves space-time cost with respect to the current allocation. A zero $\bar{\Delta}$ does not change the space-time cost if its interval is assigned to be occupied. As the current allocation is changed from the out starting allocation, a $\bar{\Delta}$ can never increase in value. Changing the out starting allocation implies changing vacated intervals to be occupied intervals. This change causes $|\bar{F}|$'s to decrease and $|\bar{P}|$'s to increase. Thus Δ's can only decrease. Thus nodes with nonpositive node weights can be removed from the $\bar{G}$-graph to form an optimal allocation.

Consider removing a negative weight node, y, from the $\bar{G}$-graph. When y is removed from $\bar{G}$, every node connected to y through a single edge has its node weight reduced by R. Consider a node z and assume an edge (y,z) exists. If $y \in |\bar{F}_z|$, $|\bar{F}_z|$ is reduced by one since y is assigned to the occupied state. Then $\bar{\Delta}_z$ is reduced by R. If $z \in |\bar{F}_y|$, occupying y causes $|\bar{P}_z|$ to increase by one. $\bar{\Delta}_z$ is again decreased by R. In general, if a subgraph $\bar{G}'$ is removed, for each edge $e \in (N_{\bar{G}'}, N_{\bar{G}-\bar{G}'})$, the node in $\bar{G}-\bar{G}'$ connected to e has its weight reduced by R. Theorem 3.1.1 is applicable to subgraphs of a minimum subgraph of $\bar{G}$. Thus we can apply the RED(i) algorithm to the $\bar{G}$-graph.

Theorem 3.3.1 The G-graph and the $\bar{G}$-graph have the same nodes and the same edges before reduction procedures are performed. The edge weights are identical. However, the node weights differ in general.

Proof: By definition, the $\bar{G}$ and G graph contain only nodes representing exactly the same nonreference intervals. Also, the edges in each graph are identical. Consider two nodes A_i and B_j in N_G. There is an edge between A_i and B_j if and only if $A_i \in P_j$ or $B_j \in P_i$. Consider the corresponding nodes in $\bar{G}$. There is an edge between A_i and B_j in $\bar{G}$ if and only if $A_i \in \bar{F}_j$ or $B_j \in \bar{F}_i$. However, if $A_i \in \bar{F}_j$, then $B_j \in P_i$, and if $B_j \in \bar{F}_i$, then $A_i \in P_j$. Thus the edges of G and $\bar{G}$ are the same. From the development of the node weights for G and $\bar{G}$ (Δ and $\bar{\Delta}$), it is clear that they differ in general.

□

Now we can apply the RED(i) algorithm to both the G and $\bar{G}$-graphs concurrently. Suppose that in the concurrent application of RED(i) to both graphs, we choose to have the set of nodes in the minimum subgraph of G to be disjoint from the set of nodes in the minimum subgraph of $\bar{G}$. In other words, we choose to have the optimal allocation derived from the minimum subgraph of G be identical to the optimal allocation derived from the minimum subgraph of $\bar{G}$. Suppose that we have detected a subgraph of a minimum subgraph of G. The nodes in this subgraph can be vacated in the optimal allocation. Thus, this subgraph can be removed from G with the RED(i) algorithm. Consider the subgraph of $\bar{G}$ corresponding to this subgraph. The nodes of the corresponding subgraph of $\bar{G}$ are currently assigned to the vacated state. Otherwise, if a node in this corresponding subgraph in $\bar{G}$ has been assigned to the occupied state, the nodes of the minimum subgraph of G and $\bar{G}$ would not be disjoint. Therefore, the set of nodes in the corresponding subgraph of $\bar{G}$ are assigned to the vacated state in the optimal allocation. Since the state of these nodes is known,

they can be removed from the $\bar{G}$ graph. However, in removing this corresponding subgraph from $\bar{G}$, none of the remaining nodes in $\bar{G}$ have their node weight adjusted. The $\bar{\Delta}$'s of the nodes remaining in the $\bar{G}$ graph are computed with the assumption that the removed nodes are vacated. Thus, the $\bar{\Delta}$'s of the remaining nodes are not changed by removing these nodes. Similarly, if a subgraph of a minimum subgraph of $\bar{G}$ is removed by the RED(i) algorithm, the corresponding subgraph in G is removed without adjusting the node weights of the nodes remaining in the G graph.

It is advantageous to apply the RED(i) algorithm to both G and $\bar{G}$ graphs concurrently. First, after the reductions are completed, there will most likely be fewer nodes whose state is undetermined. Secondly, since we can remove subgraphs detected in one graph from the other graph, each graph reduction saves unnecessary steps for checking subgraphs that will not be in their minimum subgraph. We propose the following graph definition for concurrent application of RED(i) to the G and $\bar{G}$ graphs.

Definition 3.3.1 An Augmented G-graph, G_A, is the graph containing the nodes and edges of G before any subgraph reductions are performed. Each node in G_A, has two node weights: the node weight of the corresponding node in the G graph, and the node weight of the corresponding node in the $\bar{G}$ graph.

The RED(i) algorithm can be applied concurrently to the G and $\bar{G}$ graphs by applying the algorithm to the G_A graph as follows. For the node weights of G and $\bar{G}$, subgraphs of minimum subgraphs are searched for using only the node weights associated with one of the graphs (G or $\bar{G}$) at a time. When a subgraph of a minimum subgraph is detected using node weights from say $\bar{G}$, the subgraph is removed as in the algorithm and only the $\bar{G}$ node

weights are adjusted as in the algorithm. Note that this has the effect of removing the corresponding nodes from G without adjusting node weights in G. When RED(i) is applied to the G_A-graph in this manner, the number of steps performed for detecting subgraphs of minimum subgraphs is, in the worst case, multiplied by a factor of 2 over applying it to the G-graph alone. Thus, the worst case complexity for RED(i) applied G_A is the same as applying it to G, namely $\mathcal{O}(DP*N^i)$.

After the reduction process is completed for the G_A graph, the FG graph is formed from the remaining nodes and edges of either G or $\bar{G}$. Then the maximum flow in the FG graph determines the minimum subgraph of the graph used.

In Figure 16 we perform RED(1) on the G_A graph from the example of Chapter 2. In the figures the node weights without brackets are $\bar{G}$ weights. The bracketed weights are G weights. In Figure 16a node A_1 is removed since its G node weight is negative. Only node B_2's G node weight is modified. The G weight of C_3 could be modified, but it is also removed since its $\bar{G}$ weight is negative. Node A_4 also has a negative $\bar{G}$ weight. Only the $\bar{G}$ weights of nodes adjacent to C_3 and A_4 are adjusted.

In Figure 16b, node C_5 is removed. After the reductions we are left with the same nodes as in the reduction example of Figure 7. However, if we apply RED(2), the state of the remaining nodes is obvious. Nodes B_2 and C_7 are vacated because they form a subgraph of G which fulfills conditions (i) and (ii) of Theorem 3.1.1. The G-graph weight of nodes B_2 and C_7 are .1R and .2R respectively. The weight of the subgraph containing B_2 and C_7 is .1R + .2R - R = -.7R. Nodes A_6 and B_8 are occupied because

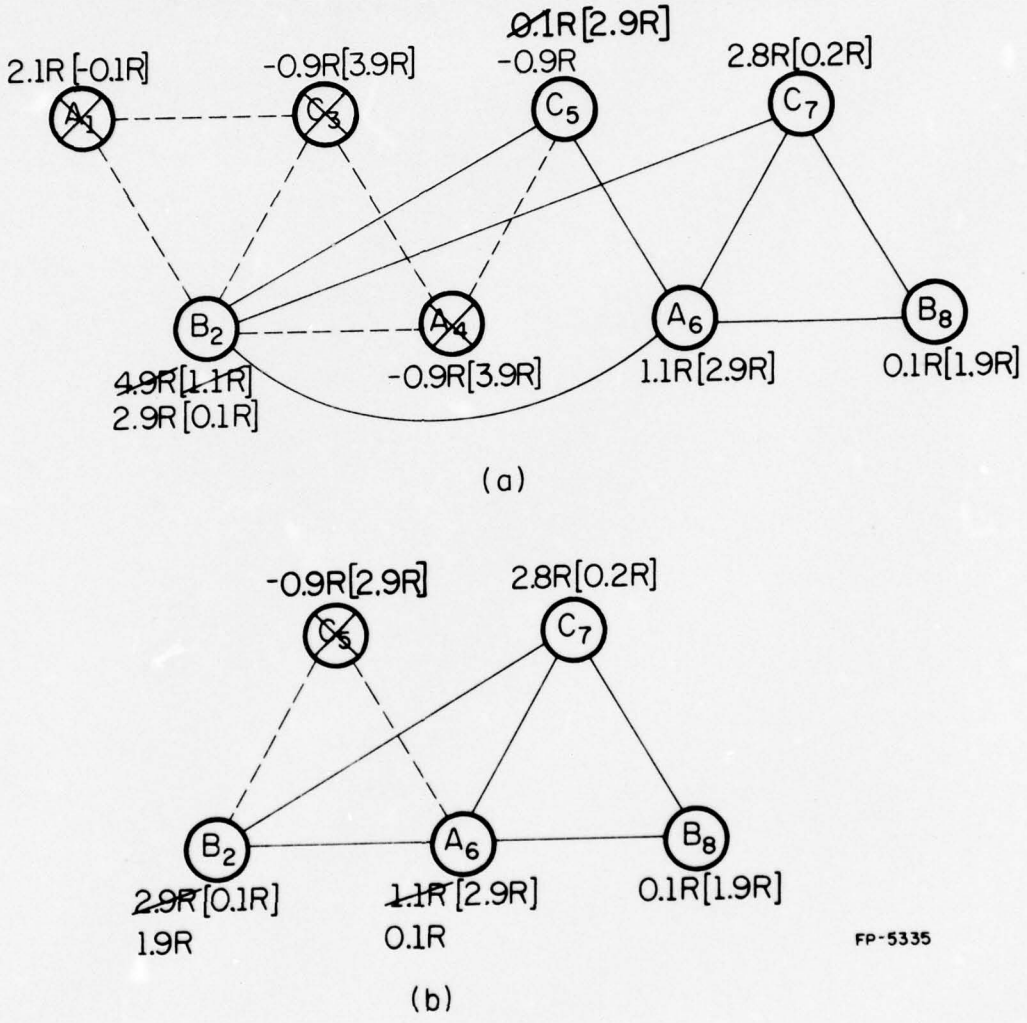


Figure 16. G_A-graph reduction.

they form a subgraph of $\bar{G}$ which fulfills conditions (i) and (ii) of Theorem 3.1.1. The weight of the $\bar{G}$ subgraph containing A_6 and B_8 is .1R + .1R - R = - .8R. Note that this assignment matches the solution in Figure 10.

3.4. Complexity Reductions for Suboptimal Allocations

The methods of sections 3.2 and 3.3 are applied to reduce the number of nodes for constructing the FG graph. In applying the maximum flow algorithm, the worst case complexity is $\mathcal{O}(N^3)$. The RED(i) algorithm attacks the complexity problem by reducing N. Applying RED(i) can achieve a further reduction in complexity. In removing subgraphs from G_A, it is possible to form disjoint subgraphs. Substantial savings in complexity can be gained with occurrence of disjoint subgraphs. In the following theorem we show the complexity for the occurrence of j disjoint subgraphs.

Theorem 3.4.1 In finding the maximum flow in a G graph (or $\bar{G}$ graph) with N nodes, $\mathcal{O}(N)$ edges, and j disjoint subgraphs, the worst case complexity for Dinic's algorithm is $\mathcal{O}(j^* \max N_k^3)$ where N_k is the number of nodes in the k^{th} disjoint subgraph and $\sum_{k=1}^{j} N_k = N$.

Proof: Consider the k^{th} disjoint subgraph. It has N_k nodes and $\mathcal{O}(N_k)$ edges. Thus, the worst case complexity for Dinic's algorithm is $\mathcal{O}(N_k^3)$. Therefore the worst case complexity for the j disjoint subgraphs is:

$$\sum_{k=1}^{j} \mathcal{O}(N_k^3) \leq j^* \mathcal{O}(\max N_k^3) = \mathcal{O}(j^* \max N_k^3) .$$

□

With the reduction methods of the previous section, the number and size of any disjoint subgraphs depend on the results of the reduction

algorithms and are as such unpredictable in general. However, after applying the RED(i) algorithm, further complexity reduction can be achieved by separating the resulting graph(s) into smaller disjoint subgraphs. The price for separating graphs is a suboptimal allocation. We determine the worst case error introduced by removing edges to separate graphs.

Theorem 3.4.2 The suboptimal space-time cost differs from the optimal space-time cost by at most $\pm (k/2)*R$ where k is the number of edges removed to form disjoint subgraphs.

Proof: The minimum weight subgraph has weight equal to the maximum flow minus R times the number of edges in G (or $\overline{G}$). Consider an edge, e, which is removed to form disjoint graphs. Form the FG graph without removing e. When FG is formed, e is cut and the cut ends are connected to a link node. The link node is connected through a source edge to the source. If the associated link node and its incident edges are removed, the maximum flow is decreased by at most R. The maximum flow decreases by the net flow into (or out of) the removed link node. The net flow into a link node cannot exceed R. There are three edges with flow capacity R incident on a link node. If the net flow into a link node exceeds R, then two of the edges must be carrying flow into the link node. However, the remaining edge can only carry a flow of at most R out of the link node. Since a link node is not a source or a sink, the net flow into a link node must equal the net flow out. Thus, the net flow in or out cannot exceed R. Therefore, by removing an edge, e, from G, the maximum flow in FG can have from no change in flow to as much as a decrease of R in flow. Similarly, if k edges are removed, the maximum flow in FG can

have from no change in flow to a decrease of k* R in flow.

Consider a flow graph, FG', which is formed from a G-graph that has k edges removed to form disjoint subgraphs. We compute the weight of the "minimum" subgraph of FG', $WT(FG'_M)$, by subtracting from the maximum flow the weight of all the edges of G except for the removed edges for which we subtract half the weight. Then the weight of the "minimum" subgraph is

$$WT(FG'_M) = MXFLO(FG') - |N_G|*R + (k/2)*R$$

where MXFLO(FG') is the maximum flow of FG'. The weight of the minimum subgraph of FG is:

$$WT(FG_M) = MXFLO(FG) - |N_G|*R.$$

If the MXFLO(FG) equals the MXFLO(FG') then $WT(FG_M) - WT(FG'_M) = - (k/2)*R$. If the MXFLO(FG) minus the MXFLOW(FG') is k*R, then $WT(FG_M) - WT(FG'_M)$ $= (k/2)*R$. Therefore since MXFLO(FG) - MXFLO(FG') varies between 0 and k*R, the error caused by removing k edges to form disjoint subgraphs is at most $\pm$ (k/2)*R.

□

3.5. An Implementation of the DMIN Algorithm

The DMIN algorithm is implemented with two programs written in the SAIL language and executed on a DEC PDP 10. The first program computes the Δ and $\overline{\Delta}$ data, and then implements a RED(1) algorithm on the G_A-graph. The second program implements Dinic's maximum flow algorithm.

In our implementation of the RED(1) algorithm, we avoid forming a graph. Instead we form edges as needed. Consider the following definition:

Definition 3.5.1 During a reduced trace scan, nonreference interval, A, is said to be active while scanning elements in the reduced trace which are within A.

Note that up to DP intervals may be active at once. An edge is formed between nodes if one of their corresponding intervals is in the P set of the other node. This criterion is based on the G-graph, but generates edges for both the G and $\overline{G}$ graphs. Consider, A, an active interval's corresponding node. For each interval B, which becomes active while A is active, there is an edge between A and B. Consider nonreference interval B. If it becomes active while A is active, the B interval ends either before or after the end of A. If B ends before A ends, then $A \in P_B$. If B ends after A ends, then $B \in P_A$. Thus we can create the edges in the G_A graph for the starting allocation and the out starting allocation by realizing that there is an edge between the node corresponding to an interval which has just become active and the nodes representing those intervals which are currently active during a scan of the reduced trace. By scanning the reduced trace from beginning to end, we can use this method to generate all of the edges in G_A.

The data structure for the reduction implementation is a matrix with 4 rows and N columns where N is the number of elements in the reduced trace. One row contains the reduced trace, two of the other rows contain separately a list of the differential costs and a list of the out differential costs. A nonreference interval begins after each element in the trace. We choose to have each nonreference interval's associated Δ and $\overline{\Delta}$ in the same column as the trace element which occurs just after

the end of the interval; i.e., the next reference to the page associated with the interval. The 4th row is used for storage during scans. .

The Δ and $\overline{\Delta}$ data is computed in a forward scan. We scan the memory reference trace to form the reduced reference trace as we compute each interval's Δ and $\overline{\Delta}$. We can compute and store an interval's Δ and $\overline{\Delta}$ in the following way. Each time we encounter the beginning of non-reference interval, we count the number of occurrences within the interval of the following three quantities based on the two starting allocations: the number of page faults, the number of memory references, and the number of reduced trace elements. As we scan in the forward direction, we record the number of pages in the memory resulting from the starting allocation and the space-time cost resulting from the starting allocation. When we reach the end of interval i, the interval's Δ_i is computed with the following available information: the number of page faults occurring for the starting allocation,to form $|F_i|$; the number of memory references within the interval,to form τ_i; and the number of pages currently in the memory under the starting allocation, to form $|P_i|$. Then $\Delta_i = (2 + |P_i|)*R - (\tau_i + |F_i|*R)$. We compute $\overline{\Delta}_i$ with the following: the number of memory references,to form τ_i; and the number of reduced trace elements occurring within the interval, to form $|\overline{F}_i|$ under the out starting allocation. Then $\overline{\Delta}_i = (\tau_i + |\overline{F}_i|*R) - 2*R$. Note that $|\overline{P}_i|$ is zero under the out starting allocation. The Δ_i and $\overline{\Delta}_i$ computed for the i^{th} interval are stored in the same column as the most recently encountered reduced trace element, i.e., the trace element which occurs just after the end of the i^{th} nonreference interval.

This method for computing the Δ and $\overline{\Delta}$ data, i.e., the interval data, is completely sequential, i.e., no backtracking is required for computation or storage with respect to the matrix containing the reduced trace.

In implementing our RED(1) algorithm, it is impractical to have an array in the program's address space with as many columns as there are reduced trace elements. Our solution to this problem is to implement an automatic overlaying procedure under program control. We used an array with 4 rows and 4096 columns. As we scan the memory reference trace to compute the interval data, we store the data in the 4096 column array. When the data would overflow the array, we write the array data onto a disc file and then restart storing new interval data in this array. Since the interval data is completely sequential with respect to the columns in the array containing the entire reduced trace (which is stored on a disc) we do not need multiple arrays. Also, the reduction phase is sequential in the same sense. Thus, during the entire RED(1) execution, a single array is sufficient. Furthermore, since the array containing the entire interval data is stored on a disc, the size of the reduced trace is limited only by the amount of available disc space.

Once the reduced trace interval data is computed, the reduction phase begins. The reduction phase is identical to the RED(1) procedure although the order of the steps differ. Our implementation varies in one respect. We don't maintain a list of edges because this could increase the amount of disc space needed by a factor of about DP. Each time we remove a node, we generate the edges from the removed node to the other nodes in G_A. In order to make the process completely sequential with

respect to the interval data (reduced trace, $\Delta,\overline{\Delta}$), we accomplish edge removal and node adjustments in a backward scan followed by a forward scan, i.e., a dual scan.

In the backward scan, we search for negative weight nodes (of G and $\overline{G}$). Suppose that we have found that node A in G has negative weight. We check the node weights at the end of interval A, the place in the trace where the data is stored. Those intervals which become active while A is active have edges from their nodes to A's node. When such intervals become active, their G and $\overline{G}$ node weights are available. Thus, the node weights can be adjusted and checked when they become active. It is possible that several nodes from both graphs are being removed concurrently. Thus, when a node becomes active, it may be possible to subtract several edges from it. We use a variable which is equal to the number of edges to be subtracted from each type of node weight. This variable is updated as follows. Since A is to be removed from the G graph, when A becomes active, the number of edge weights to be subtracted from G type node weights is increased by one. When A becomes inactive, the number to be subtracted from G type nodes is decreased by one. A similar variable update method is used for $\overline{G}$ type node weights.

Those intervals which are active when we determine that the node A is to be removed have edges from their corresponding nodes to interval A. However, since we are scanning backwards, we have already passed the end of the intervals which are currently active, i.e., the position where such interval's data is stored. Thus, if we are to adjust the node weights of the nodes of the active intervals, we would have to go through the

interval data nonsequentially. Going through the interval data nonsequentially would substantially increase I/O since we might have to read in an entire array of data from the disc just to adjust one datum. We can avoid the nonsequential data references in the following way.

When we need to subtract edge weights from node weights of intervals that have become active before we know of the subtractions, we store the number of edges which are to be subtracted from each such interval's two types of node weights. When such an interval becomes inactive, we store the number of edges to be subtracted from each type of node weight. This data is stored in the column of the matrix which contains the reduced trace element which marks the start of such an interval. The fourth item in the column stores the number of edges to be subtracted from the $\overline{G}$ node weight. The number of edges to be subtracted from the G node weight shares the first row with the reduced trace element. For the PDP 10, integer variables have about $10\frac{1}{2}$ decimal digits. The three least significant decimal digits contain the page name of the reduced trace element. The number of edges to be subtracted from the G node weight is multiplied by 1000 and added to the page name, this scheme allows up to 1000 distinct pages. It further allows for more than 35 million edges to be subtracted from a node weight. If the limits outlined above are exceeded, an overflow occurs and the program halts. The number of edges to be subtracted are stored with this method until the next forward scan. In the forward scan, each column is checked to see if any edges are to be subtracted. The number of edges to be subtracted for each type of node weight is stored for such an interval

until the end of the interval. At this point the node weights are available to be adjusted and checked.

Each time an adjusted node becomes negative, there is a new node to be removed. During each dual scan (backward scan followed by a forward scan), we count the number of new nodes to be removed. If there is a least one node left to be removed, a new dual scan begins. Otherwise, the reduction phase is completed.

In Table 2 we list the performance results for our RED(1) implementation. In the first two columns we list the experimental parameters. We use two programs, two page sizes and three reactivation times. These parameters are discussed in chapter 4. In the third column we list the number of memory references in the trace used in the reduction program. In the fourth column, we list the number of nodes with an undetermined state after the reduction is completed. In the "ITERATIONS" column, we list the number of dual scans performed in the reduction phase. In the "DISJOINT GRAPHS" column, we list the number disjoint graphs occurring from the reduction process.

In general, our RED(1) implementation performed well except for LIST/512 with a 5000 memory reference reactivation time. For this execution, the reduction program runtime became excessive and was not run to completion. The data in Table 2 for this run is the result of about 7 hours and 40 minutes of CPU time. The longest runtime among the other runs is about 1 hour and 42 minutes for LIST/4096 with a reactivation time of 500.

Table 2

RED(1) Performance Results

PROGRAM/ PAGE SIZE	REACTIVATION TIME	TRACE REFERENCES	NODES AFTER REDUCTION	ITERATIONS	DISJOINT GRAPHS
GAUSS/4096	50	83,377	0	3	-
GAUSS/4096	500	83,377	16	4	2
GAUSS/4096	5000	83,377	2	6	1
GAUSS/512	50	83,377	272	20	16
GAUSS/512	500	83,377	5	8	1
GAUSS/512	5000	83,377	8	6	2
LIST/4096	50	499,413	11,279	7	146
LIST/4096	500	499,413	305	24	11
LIST/4096	5000	499,413	0	11	-
LIST/512	50	145,134	16,098	13	51
LIST/512	500	145,134	12,803	106	1
LIST/512*	5000	145,134	43,233	142	1

*Not run to completion

After the reduction algorithm is completed, the nonreference intervals remaining are used to generate a flow graph from the G-graph using the methods described above. We have implemented Dinic's maximum flow algorithm [8,11]. Dinic's algorithm starts with a breadth-first search. A breadth-first search involves a labeling process. Each node in the graph is in one of three states: labeled, labeled and unscanned, or labeled and scanned. Initially all nodes are unlabeled. The search starts at the source. The source is labeled zero. All link nodes are labeled 1 if their edge from the source is nonsaturated. Such nodes are reachable. After all link nodes reachable from the source are labeled, the source is said to be scanned. A node is scanned if all nodes reachable through a single nonsaturated edge are now labeled. The labeling process continues by scanning nodes in the order they are labeled, i.e., first-labeled-first-scanned. When scanning a node labeled n, all unlabeled nodes are labeled 1 + n if they are reachable through a single edge from the node being scanned. We continue the labeling process until the terminal node is labeled with, say f. The labeling process continues until a node with label f is to be scanned. This breadth-first search yields all minimum length flow augmenting paths. As the FG graph is scanned, we note the nodes and edges traced. These nodes and edges form an <u>auxiliary graph</u>.

After the breadth-first scan is completed, a series of depth-first searches are performed on the just formed auxiliary graph. A depth-first search traces flow augmenting paths from the source to the terminal. The trace starts at the source and continues through a useable edge to a node labeled 1, and then through a useable edge to a node labeled 2, etc.

The path is traced until the terminal node is reached. The edge in the path with the least flow capacity determines the increase in flow. Flow is pushed through the path until the edge with least capacity is saturated. The saturated edge is removed from the auxiliary graph. Another depth-first search then begins. If in the search we proceed to a node which has no useable edge leaving the node, we backup to the preceding node in the path and delete the last edge from the auxiliary graph. The depth-first search continues until we backup to the source and no useable edge leaving the source exists. After completion of the depth-first search, another breadth-first search is commenced. If in the course of a breadth-first search, the terminal node is not reached, the flow algorithm terminates since no more flow can be pushed from the source to the terminal. The nodes in the auxiliary graph for the last breadth-first search are the nodes in the minimum subgraph.

In our implementation of Dinic's algorithm, we start the algorithm with an initial flow. Before forming the FG graph, each edge in G is connected to two nodes in G. The nodes in G are connected to the terminal node through edges called terminal edges. When we cut an edge in G to form two link edges, we proportion the initial flow among the two link edges according to the capacity of each node's terminal edge. If one terminal edge is saturated, we push as much flow as possible through the other terminal edge. Our implementation of the flow algorithm is designed to accommodate any graph whose data can fit in the available disc space.

CHAPTER 4

EXPERIMENTS WITH DMIN

The purpose of this chapter is to describe the memory allocation experiments performed and discuss their results. In the experiments, we use memory reference traces gathered from programs run on an IBM 360/75 computer. We compare the memory allocation performance of the DMIN algorithm with three other allocation algorithms: Belady's MIN algorithm, Chu and Opderbeck's Page Fault Frequency algorithm (PFF), and Smith's Damped Working Set algorithm (DWS).

4.1 Experimental Description

We use memory reference traces from two programs. One program, GAUSS, performs a Gaussian Elimination on a 14 x 14 matrix to solve a set of coupled equations, the program is written in Fortran IV. The GAUSS program is a numerical program. The other program, LIST, performs list processing. The LIST program first builds list structures. After the list building phase, the tree structure of the data is examined. For each sublist head, the set of all accessible nodes is determined. The LIST program is written in PL/1.

The DMIN algorithm is written in SAIL and is run on a DEC PDP 10. First RED(1) is executed. Then an implementation of Dinic's maximum flow algorithm [9] determines the optimal allocation. For each of the two traced programs described above, the optimal allocation is computed for several page sizes and reactivation times. The MIN algorithm, the PFF algorithm, and the DWS algorithm are used to determine allocations for the same program traces, page sizes and reactivation times.

We implement the MIN algorithm using the method of Mattson et al. [13]. First a backward scan of the reduced trace is performed to find the forward distance between references to the same page. Secondly, a forward scan is performed. In the forward scan for a certain size of allocation, whenever it is necessary to replace a page, the MIN replacement rule is to push from among the pages in the current allocation, that page which is referenced furthest in the future. We apply this rule to all sizes of allocations concurrently. Thus, we compute the number of page faults and the space-time cost for all possible allocation sizes, i.e., allocation sizes of one page up to the number of distinct pages referenced in the trace.

Another allocation algorithm used is Chu and Opderbeck's Page Fault Frequency (PFF) algorithm [6]. The PFF algorithm is a heuristic, realizable dynamic allocation algorithm. We choose to compare the PFF algorithm with DMIN because the PFF algorithm is implemented with a minimum of additional hardware, i.e., a use bit for each page of the physical primary memory. The PFF algorithm has one parameter, P, the page fault frequency parameter. Consider the inverse of P, call it T. The T parameter is a threshold value for the run time between successive page faults for a program.

The PFF algorithm makes an allocation decision only at the occurrence of a page fault. If the run time since the last page fault is greater than or equal to T, all pages not used since the last page fault are deallocated. If the run time since the last fault is less than T, none of the pages are deallocated. Two actions are performed regardless of the length of the run time since the last fault. The page needed to continue execution is transferred into the primary memory, and the use bits of each page are set to the not used state.

The other allocation algorithm used is Smith's Damped Working Set (DWS) algorithm [5]. The DWS algorithm is also a heuristic, realizable dynamic allocation algorithm. The DWS algorithm has two parameters: the working set parameter, W, and the multiplicative parameter, MULT. Each allocated page in the primary memory has a timer. A page's timer stores the number of memory references completed by its associated program since the last memory reference made to this particular page. After each memory reference is completed, the least recently used page's timer is checked. If the timer's value equals W, the associated page is immediately deallocated. A page can also be deallocated at the occurrence of a page fault. When a page fault occurs, the timer of the least recently used page is checked. If this timer has a value which is greater than or equal to MULT*W the page associated with this timer is deallocated. The MULT parameter is less than or equal to one. If MULT equals one, the DWS algorithm is identical to Denning's Working Set algorithm [4]. At the occurrence of a page fault, the required page is transferred into the primary memory. Each time a page is referenced, its timer is set to zero.

Each of the programs is traced on an IBM 360/75 computer using a modified version of the University of Waterloo's TRACE/360 program. The relevant output of the TRACE/360 program is used to generate a memory reference trace.

For the two programs traced, the object code and data fit within the core memory of the 360/75. The 360/75 is a batch machine without virtual storage. Each program is assigned a contiguous block of memory locations for its data and code. This block is fixed for the entire run of the program. Since the data and code are compact and are never

reallocated, it is meaningful to cut the data and code into pages. At most one page is only partially filled with code and/or data.

In these experiments we vary seven parameters. The first three apply to all algorithms. The fourth parameter applies only to the MIN algorithm. The fifth parameter applies only to the PFF algorithm. The sixth and seventh parameters apply to the DWS algorithm.

The first three parameters are program, page size, and reactivation time. As described above we use two programs: GAUSS and LIST. We use two page sizes: 512 bytes and 4096 bytes. For these programs, 4 bytes corresponds to one memory word. Thus in terms of memory words, the page sizes are 128 words and 1024 words. We use three values of reactivation time: R equals 50, 500 and 5000, where the time unit is the average time between memory references of a running program. In these experiments, we set the deallocation time, D, equal to the reactivation time. We use both programs, both page sizes and all three values of reactivation time for the MIN and PFF experiments.

The fourth parameter is the size of static allocation for the MIN algorithm. In our experiments with MIN, we used the entire range of allocation size, i.e., from one page up to DP, the number of distinct pages referenced in the trace. The fifth parameter is T, the inverse of the page fault frequency parameter. In the PFF experiments we decreased T until the space-time cost begins to increase. We did not decrease T further, since the value of T at this point was always 50 memory references or less. We increased T until the only faults occurring are attributable to the first reference to each page. Any further increase in T would have no effect on the PFF allocation.

For the DWS algorithm experiments, we used only the LIST program for comparison with DMIN. We use reactivation times of 50, 500, and 5000 memory references in these experiments. We do not perform the DWS experiments for the GAUSS program because the DMIN allocation's for the GAUSS program are close to constant. Thus, we expect the DWS algorithm to perform about as well as the PFF algorithm does. The sixth and seventh parameters of these experiments are the W and MULT parameters respectively. We choose the W parameter values based on the τ distribution of the vacated intervals from the DMIN allocation. We also base the W parameter values on the PFF experimental results. Where it is appropriate, we use three values for MULT: 1, .5, and .25. We used one as a value for the MULT parameter because this causes the DWS algorithm to be identical to Denning's Working Set algorithm. The other two values of MULT give a range over which to study the DWS algorithm.

We have chosen these page sizes and reactivation times to model current and, hopefully, future, memory architecture. In terms of today's technology [14], we expect the 128 word page and 50 memory reference reactivation time to be a good approximation for a cache-RAM memory architecture. At the other end of the spectrum, we expect that the 1024 word page and the 5000 memory reference reactivation time to be a reasonable approximation for a disc secondary memory. We expect that both CCD and magnetic bubble secondary memories will perform within these two extremes.

In these experiments we use the same reactivation times for both page sizes. For memory architectures with a small reactivation time, a large portion of the reactivation time is page transport time. Thus, for identical memory architectures, the reactivation time for the 512 byte page

may be significantly smaller than the reactivation time for the 4096 byte page. If identical architectures are assumed when comparing the results for a 512 byte page experiment with the results for a 4096 byte page experiment, the 512 byte page result should have a smaller reactivation time. Thus when we compare these results for equal reactivation times, the 512 byte page space-time cost is overestimated and the number of page faults are underestimated. This effect is most pronounced for R equals 50. For R equals 5000 the effect is much smaller. However, if one assumes that the same reactivation time for the 512 byte page and the 4096 byte page is achieved by using different architectures, e.g., different speed secondary memories, the comparison is accurate.

4.2 Experimental Results for DMIN versus MIN

In this section we describe the results of our computer experiments. First, performance is compared for the two programs under DMIN and MIN for several values of reactivation time and page size.

In comparing DMIN with MIN, a problem arises as to what size of allocation for MIN should be chosen. In Figures 17 and 18, we show the space-time cost and the number of page faults for MIN as a function of the size of allocation with the LIST program for R = 50 and a 4096 byte page. The graphs for this case are representative of all twelve cases (2 programs, 2 page sizes, 3 values of R). We see that both the space-time cost and the number of page faults vary significantly over the possible sizes of allocations.

We choose to compare DMIN with MIN for all three values of R and two page sizes for each program. Three sizes of static allocation for MIN

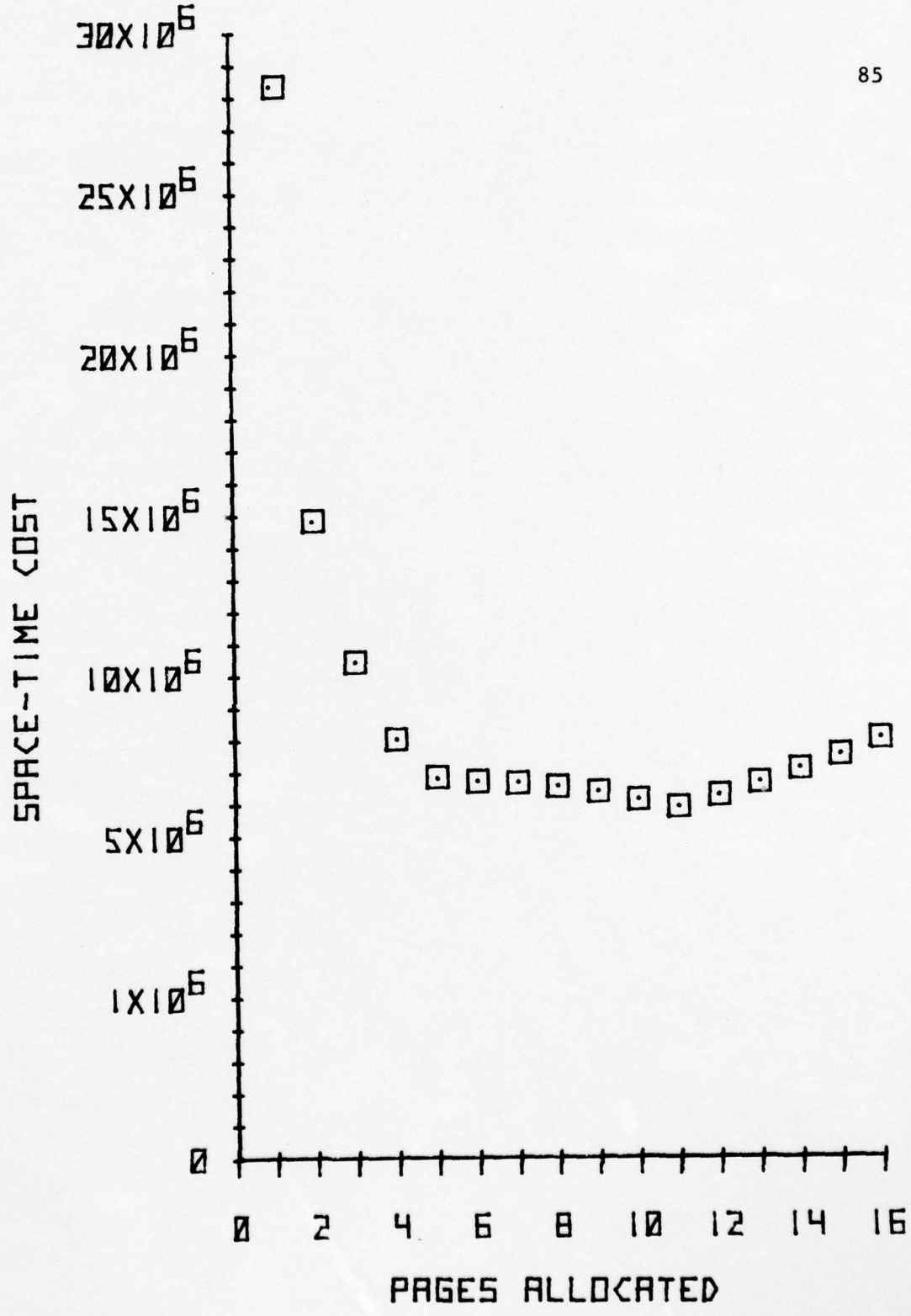


Figure 17. The MIN space-time cost for LIST with a 4096 byte page and a reactivation time of 50.

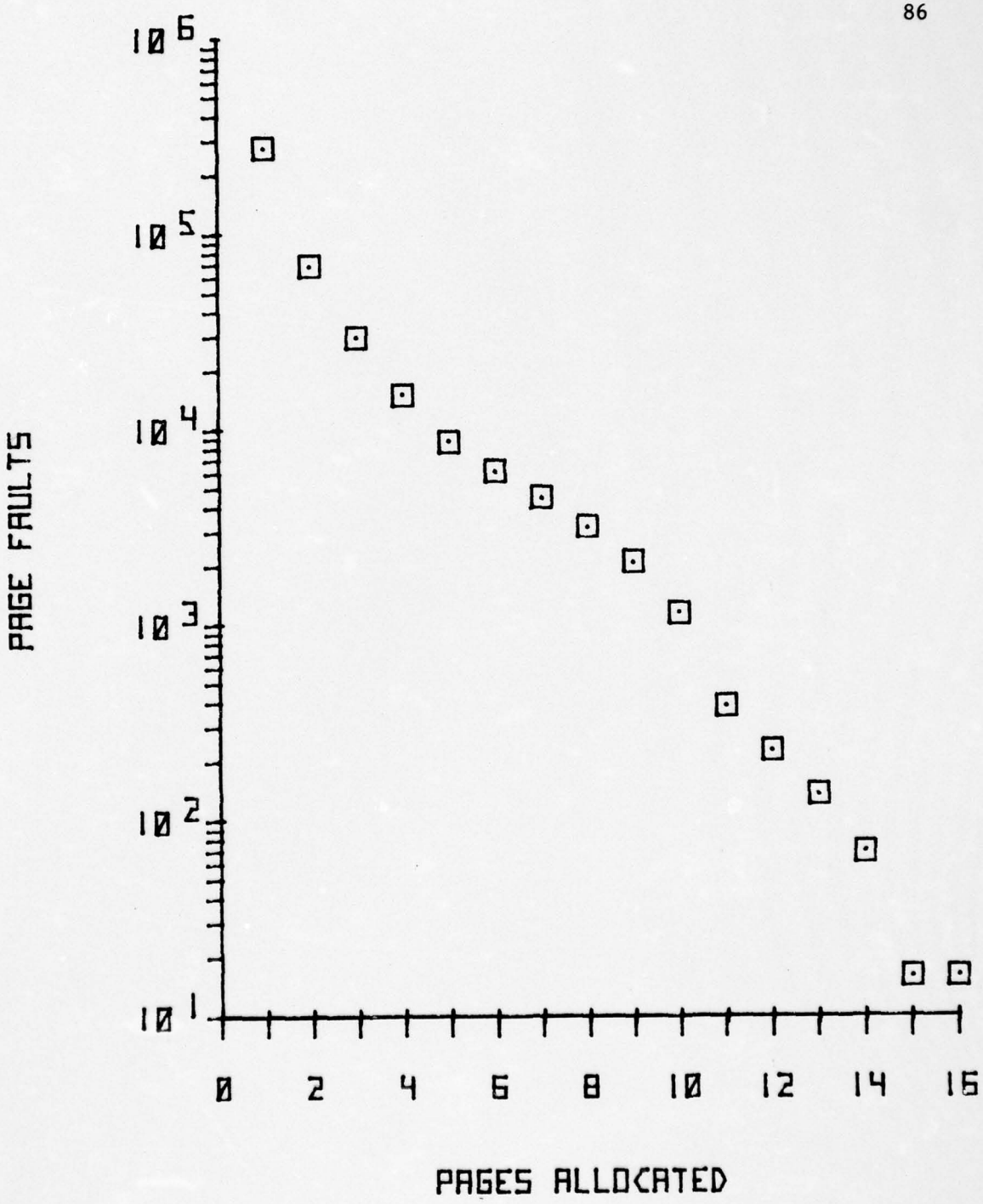


Figure 18. The page faults from MIN for LIST with a 4096 byte page and a reactivation time of 50.

AD-A040 763 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
DYNAMIC MEMORY ALLOCATION FOR A VIRTUAL MEMORY COMPUTER.(U)
JAN 77 R L BUDZINSKI DAAB07-72-C-0259
UNCLASSIFIED R-754 NL

2 OF 2
AD A040763

END

DATE FILMED
7-77

are used for each comparison. These are representative of values which could be carefully chosen with the benefit of hindsight. Other values of static allocation would be substantially inferior in space-time cost. First we compare DMIN to MIN for the allocation where MIN has the minimum space-time cost. We refer to this case as MIN(MIN). Second, we compare DMIN with MIN for the allocation size of MIN which is the rounded-off value of the average allocation size, AVG, from DMIN. We refer to this allocation size for MIN as MIN(AVG). Finally we compare DMIN with MIN for the allocation of MIN which has the number of page faults which is nearest to the number of page faults, PF, scheduled by DMIN. We refer to this case as MIN(PF). Thus, MIN(MIN) shows the best space-time cost achievable by any static allocation. MIN(AVG) shows the run time (page fault) penalty caused by matching the average allocated space of DMIN. Finally, MIN(PF) shows the space penalty caused by matching the run time of DMIN.

4.2.1 DMIN versus MIN for GAUSS

In Figure 19 we plot the space-time cost for DMIN versus: MIN(MIN), MIN(PF), and MIN(AVG) for the GAUSS program with a 4096 byte page. The space allocated to MIN for each case is shown in parentheses in Figure 20. We see that the ratio of space-time cost of MIN(MIN) to the space-time cost of DMIN increases as R increases. We conjecture that this occurs for this experiment because each time MIN schedules a fault that increases space-time cost, the penalty in space-time cost grows as R grows. For this experiment, the MIN(MIN) allocation size, 3 pages, is the same for all values of R.

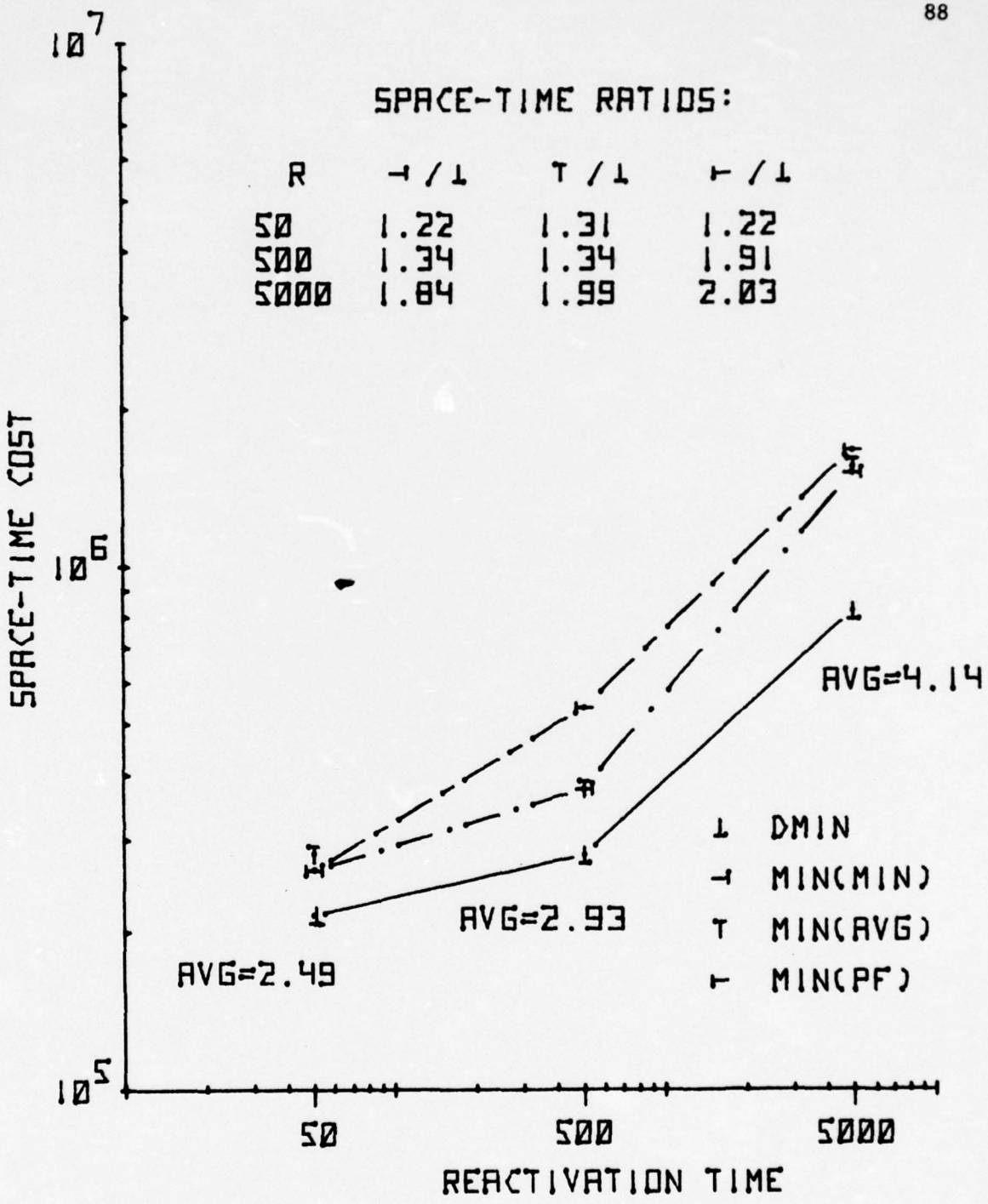


Figure 19. Space-time cost comparison of MIN with DMIN for GAUSS with 4096 byte page.

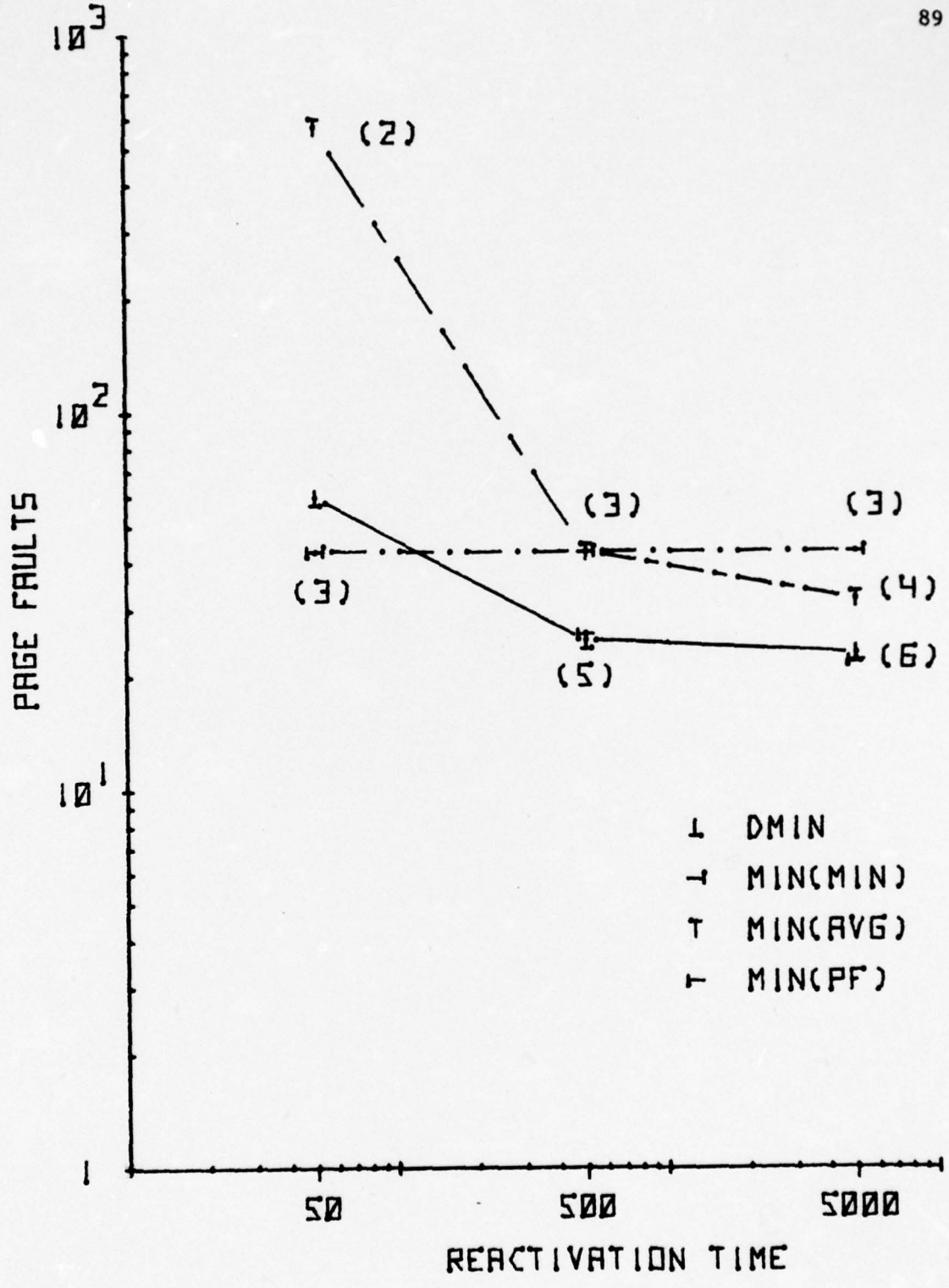


Figure 20. Page fault comparisons of MIN with DMIN for GAUSS with a 4096 byte page.

In Figure 20 the number of page faults scheduled by DMIN and the three cases of MIN are plotted. Except for R equals 50, the DMIN allocation achieves less space-time cost with fewer page faults. In Figure 20, the number in parentheses next to points plotted from the MIN allocation is the size of allocation for MIN. We see that the MIN(AVG) allocation produces both more page faults and more space-time cost than did the DMIN allocation. For the MIN(PF) allocation, the space-time cost is substantially greater, especially for the larger values of reactivation time. For R = 500 and 5000, the space-time cost of MIN(PF) is nearly double DMIN's cost.

In Figure 21 the space-time cost for the Gauss Program with a 512 byte page size is plotted. In Figure 22, the number of page faults for this experiment is plotted. The comparative results of MIN and DMIN are similar to those with the 4096 byte page. In general, MIN(AVG) will be poor in the number of page faults, while MIN(PF) will be poor in the amount of space required. In Figure 21, the space-time cost unit is normalized to the 4096 byte page size so that Figures 19 and 21 may be compared. Under DMIN, the space-time cost for the smaller page size is smaller than the cost for the larger page sizes. For R = 50, the space-time cost for the 4096 byte page size is 3.39 times greater than the cost of the 512 byte page size. For R = 500, the factor decreases to 2.46, and for R = 5000, the factor is 1.55. For this program, the smaller page size is most preferable for smaller R. The large savings in space-time cost is obtained only by a large increase in the number of page faults. For the range of R, the 512 byte page experiment has between 4 and 5 times as many page faults as the 4096 byte page experiment, but uses only 20%

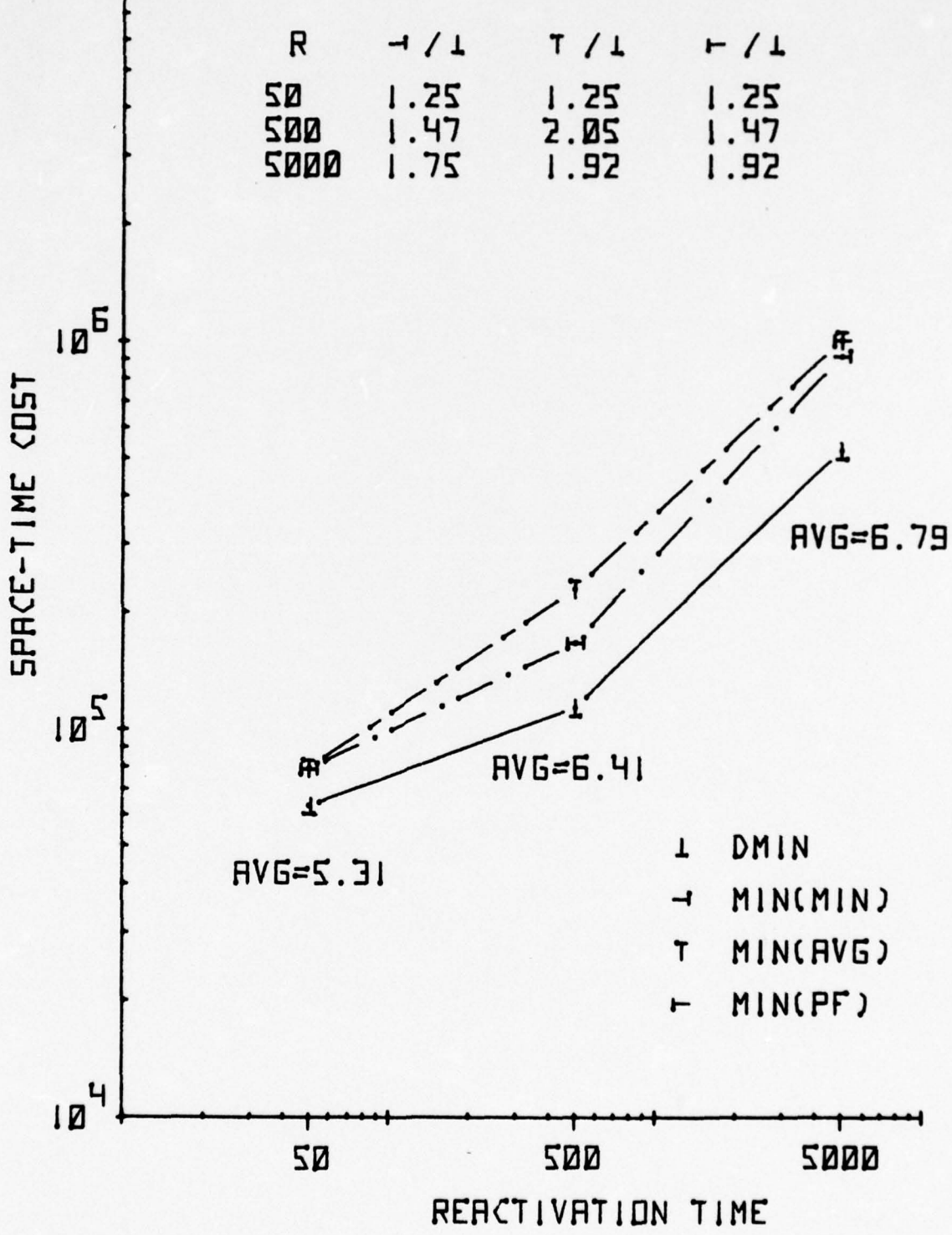


Figure 21. Space-time cost (normalized to a 4096 byte page) comparison of MIN with DMIN for GAUSS with a 512 byte page.

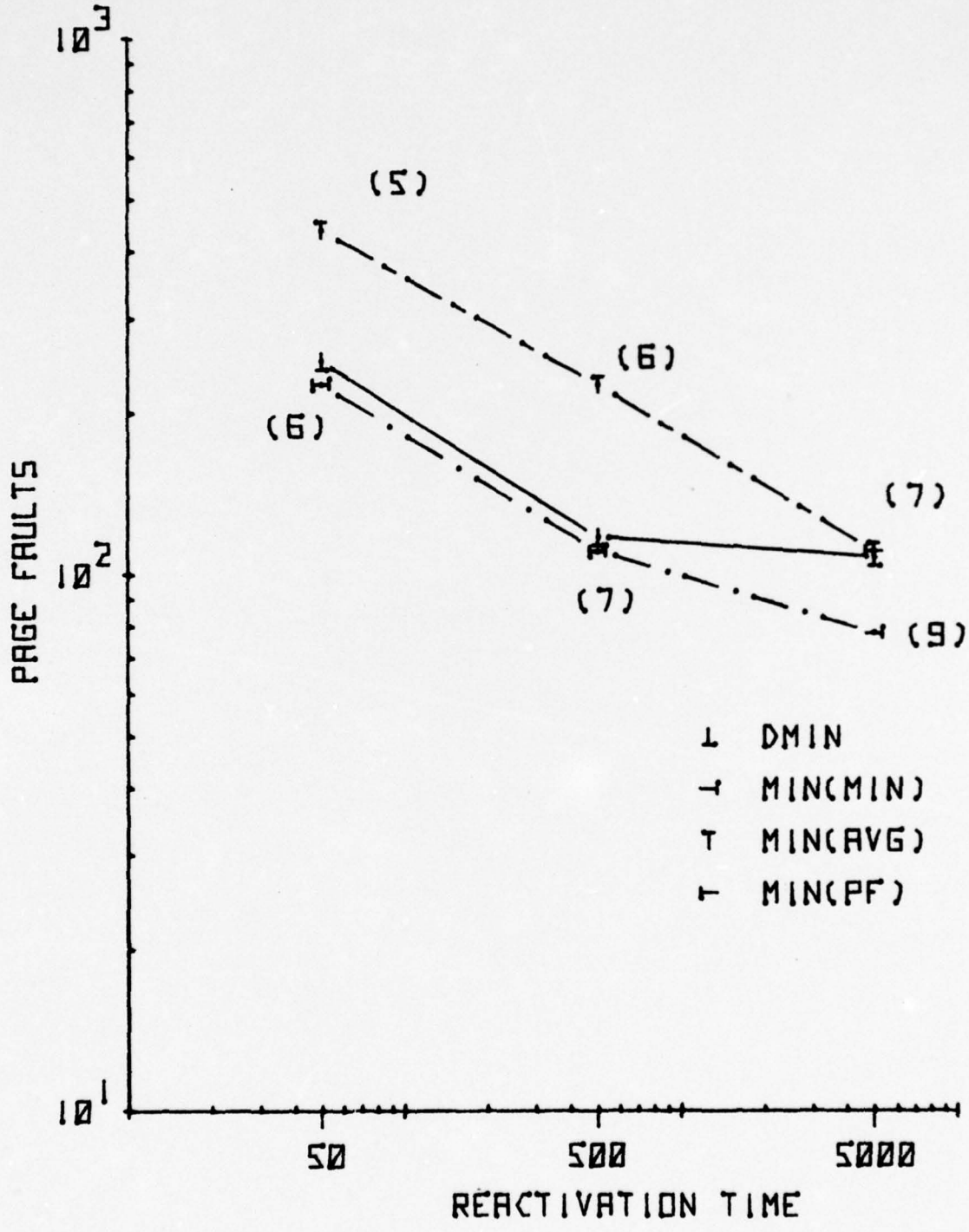


Figure 22. Page fault comparison of MIN with DMIN for GAUSS with a 512 byte page.

to 27% of the space-time cost. Since the space-time cost is smaller for the smaller page, the increase in run time, which is proportionately less than the increase in the number of page faults, is more than made up for by the decrease in allocated memory space. Another reason for the savings in space-time cost occurs because on the average, about half of the 4096 byte page is not used. The number of distinct pages referenced, DP is 11 for the 4096 byte experiment. For the 512 byte page experiment, DP is 46. However 46-512 byte pages is between 5 and 6 - 4096 byte pages. Thus, on the average about half of the 4096 byte page is not referenced.

In the previous paragraph, we see that the space-time cost ratios of the 4096 byte page experiments to the 512 byte page experiments decreases as R increases. To form the 512 byte page, each 4096 byte page is broken up into 8 smaller pages. Let A be the set of 8 smaller pages comprising one large page. Call this large page B. For the 4096 byte page experiment, consider the optimal allocation for some interval in which page B is in the primary memory. Even if only a small portion of B is being referenced within the interval, the entire B page must be in the primary memory. However with a 512 byte page, it may occur that only some proper subset of A needs to be in the primary memory for the same interval. Thus, space-time cost would be saved if only a proper subset of A is allocated memory in the optimal allocation. However, the subset of A which can be deallocated to save space-time cost is dependent upon R. From the data, we see that for DMIN, as R increases, the number of page faults decreases. However, if the number of page faults decreases, fewer pages are being deallocated during their nonreference intervals. Thus, the size

of the subset of A which could be deallocated to save space-time cost tends to decrease as R increases. Thus as R increases, the smaller page tends to save less space-time cost.

For the 512 byte page GAUSS experiment, we see that ratios of space-time cost under MIN for the various allocations to the space-time cost of DMIN are similar to those for the 4096 byte page experiment. In Figure 20, we see that for the MIN(MIN) allocation, the number of page faults for MIN is less than the number of page faults for DMIN. The difference between the number of faults, however, is small--no more than 38% (30 more page faults). For the MIN(AVG) allocation, again we see that the MIN algorithm accrues both more page faults and more space-time cost than DMIN.

It is interesting to note the sensitivity of the MIN algorithm to the size of allocation. For the case R = 50, the MIN(AVG) allocation is one page (512 bytes) less than the MIN(MIN) allocation. Yet, the number of page faults for MIN(AVG) is nearly double the number of page faults for MIN(MIN): the number of faults increases from 227 to 440.

4.2.2 DMIN versus MIN for LIST

Before discussing the experiments with the LIST program, we highlight the differences between the GAUSS and LIST programs. One difference is in the length of the reference traces. The GAUSS trace has 83,377 references. The LIST trace has 499,413 references. Another difference is that under DMIN the size of the working set of LIST varies much more than the size of the working set of GAUSS. This is shown by the memory space profiles from the DMIN allocations for GAUSS and LIST for the case in which reactivation time, R, is 50 and the page size is 4096 bytes.

These profiles are illustrated in Figures 23 and 24. Each plotted line segment is the average size of allocation for the corresponding interval. For the GAUSS profile, each interval is 2500 memory references. For the LIST profile, each interval is 5000 memory references. In both Figures 23 and 24 we have excluded the reactivation times for processing page faults.

In Figures 25 and 26 we compare DMIN with MIN for the LIST program with a 4096 byte page. We see that the space-time ratios of MIN(MIN) to DMIN for this experiment decrease with increasing R. This is the opposite of the behavior of the Gauss programs. In Figure 26, we see that the number of faults for DMIN is much greater than for MIN(MIN) for the cases R equals 50 and 500. The decrease in space-time cost ratio as R increases is due to the behavior of the page faults. For R equals 50, DMIN schedules 1425 more (a factor of 468%) faults than occur for the MIN(MIN) allocation. The DMIN allocation uses only 68% of the amount of memory that the MIN(MIN) allocation uses. For R equals 500, DMIN schedules only 142 faults, a reduction 1670 faults from the R equals 50 case. With this reduction in the number of faults, the space-time cost rises substantially. For the MIN(MIN) allocation at R equals 500, the number of faults is minimal, i.e., the only faults are those which occur for the first reference to a page. Thus for MIN(MIN) in this case, there is not any space-time cost accrued for erroneous faults as occurred for the GAUSS program experiments. For the case R equals 5000, both DMIN and MIN(MIN) have the same number of page faults, 16. Thus, even though the working set changes for LIST, DMIN can't remove any pages because the working set changes too rapidly compared to the reactivation time.

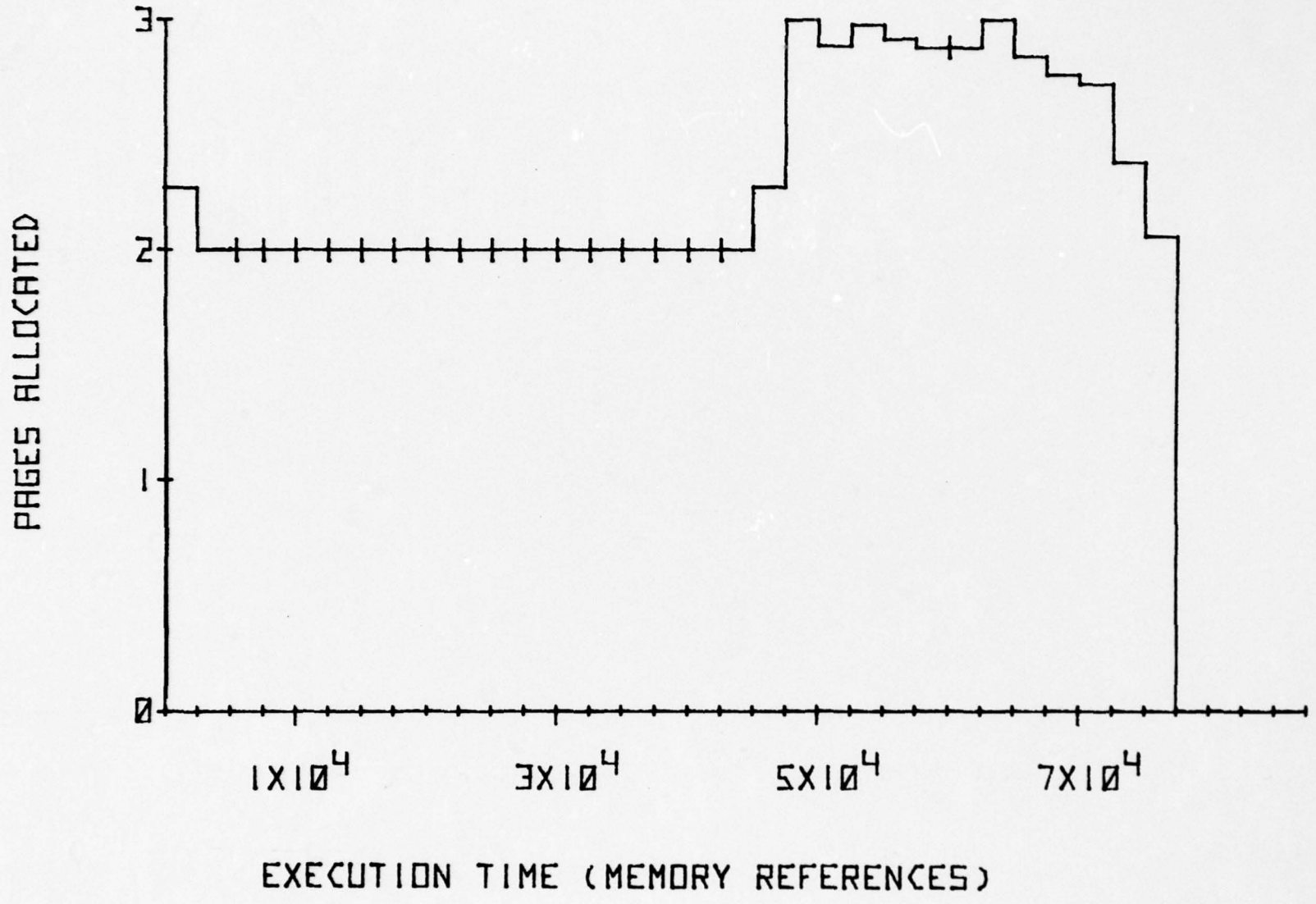


Figure 23. The average space profile from DMIN for GAUSS with a 4096 byte page and a reactivation time of 50.

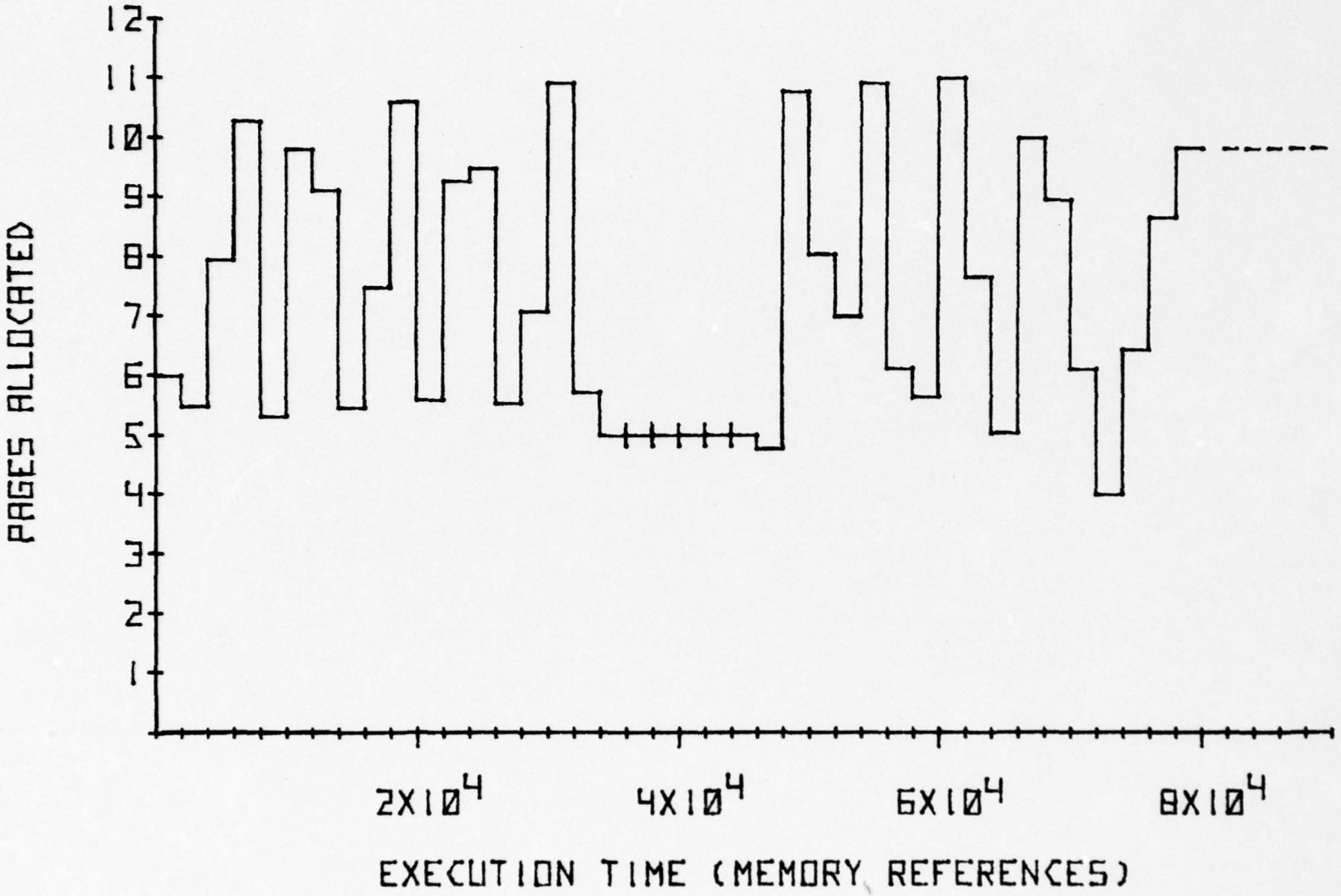


Figure 24. The average space profile from DMIN for LIST with a 4096 byte page and a reactivation time of 50.

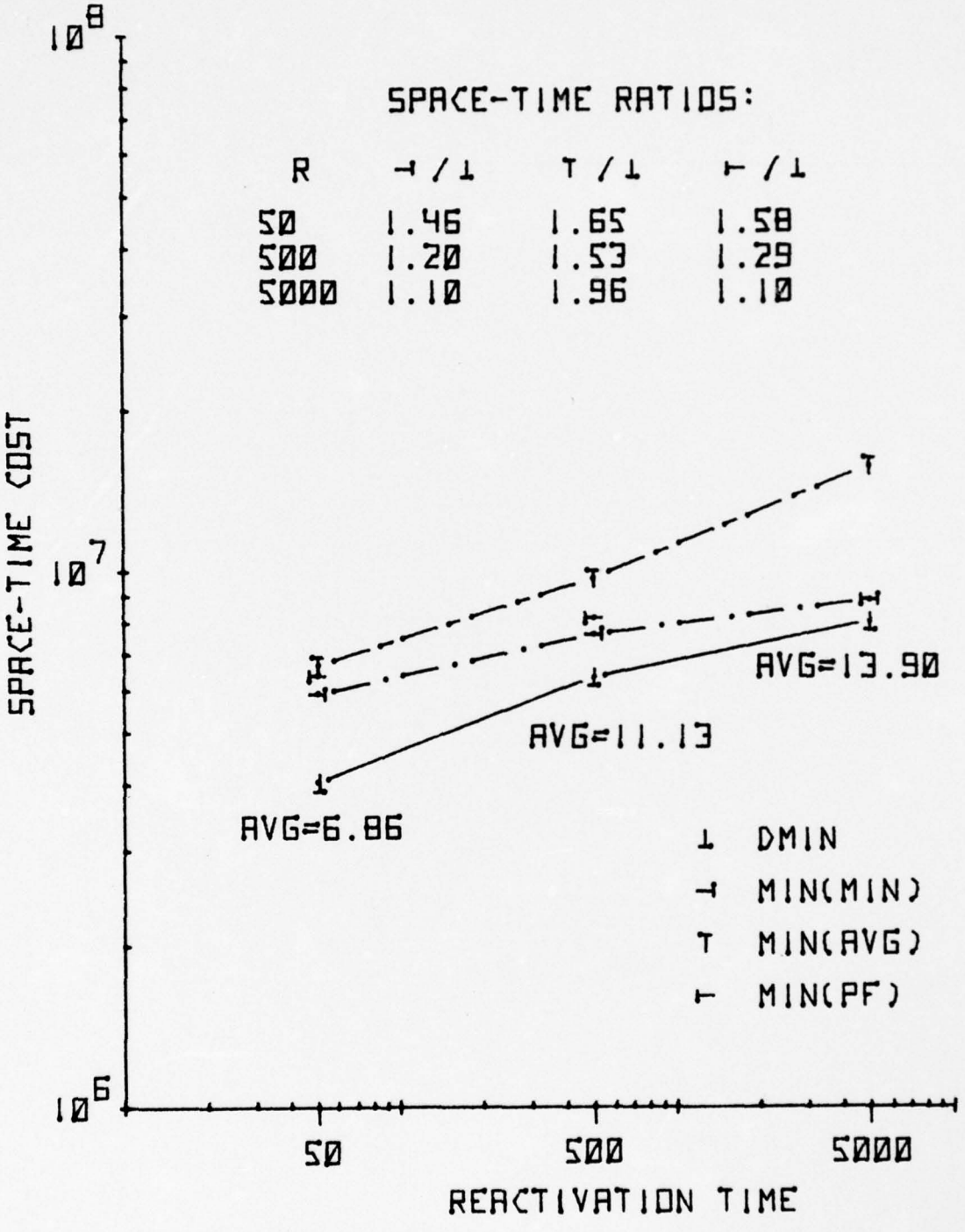


Figure 25. Space-time cost comparison of MIN with DMIN for LIST with a 4096 byte page.

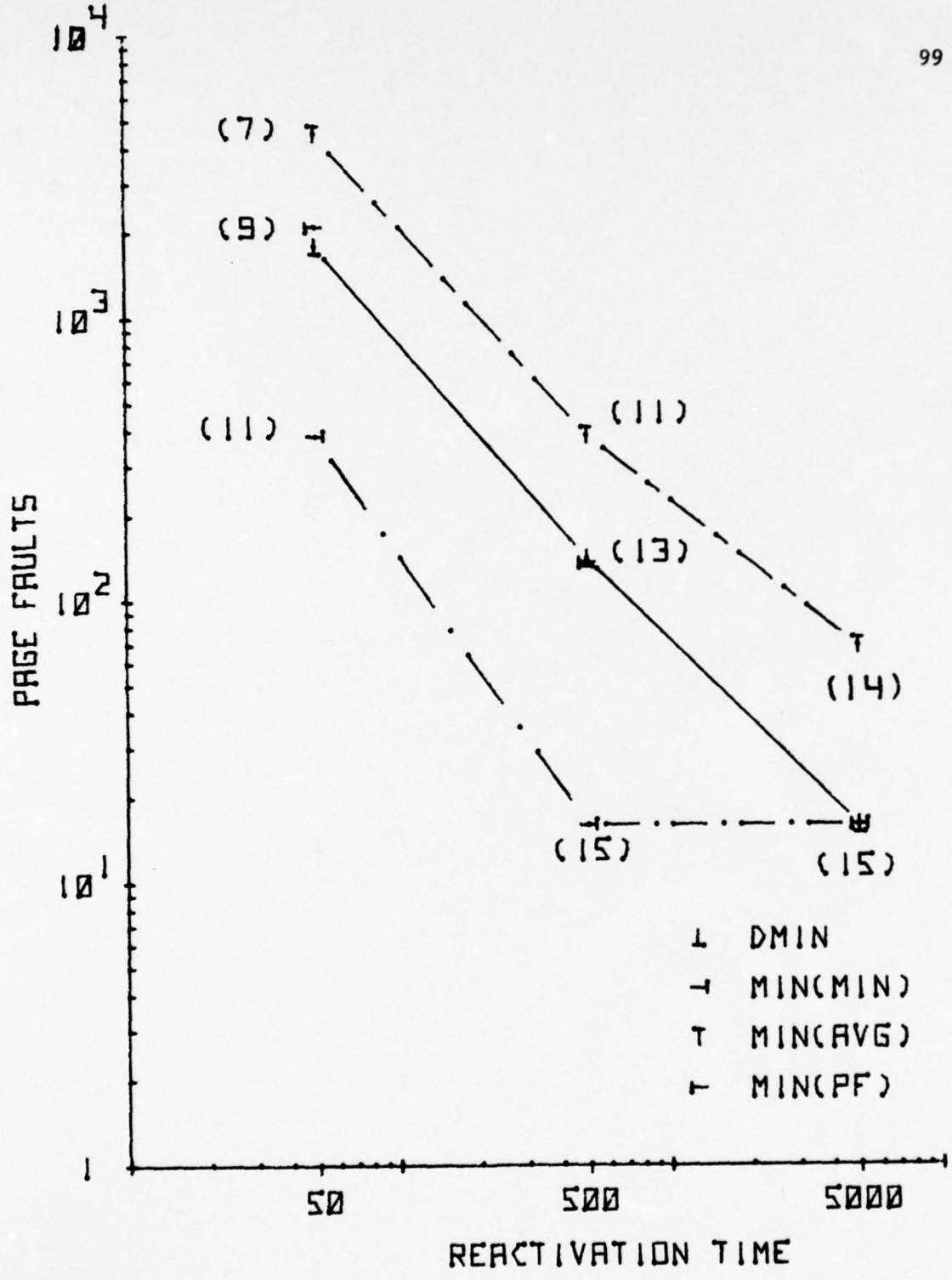


Figure 26. Page fault comparison of MIN with DMIN for LIST with a 4096 byte page.

It is interesting to note that despite having the same number of page faults, the DMIN allocation uses only 90% of the memory that the MIN(MIN) allocation does. This saving is accomplished by removing pages immediately after the last reference to the page.

For the LIST program, we see that the space-time cost for MIN(MIN) relative to DMIN is generally higher than for the GAUSS program experiments. This occurs because the DMIN allocations deviate from the average more for the LIST program than for the GAUSS program.

For the next set of experiments, we attempted to compute the DMIN allocation for the LIST program with a 512 Byte page. For this case, the computations became excessively long for the reduction process for the case R equals 5000 (greater than 7.5 hrs of CPU time on a DEC 10). For this page size, the reduced trace had 342,576 intervals. Our response to this problem was to truncate the reduced trace so that the number of intervals is 100,000. The reduction phase then become more manageable. In computing the maximum flow, the computation time became excessive for the case R equals 50. The problem here is that there are more than 550,000 edges and more than 12,000 nodes in the reduced flow graph. In applying Dinic's algorithm, the program became I/O bound. Our response to this problem was to settle for bounding the space-time cost for the optimal allocation. For the case R equals 50, we stopped the maximum flow program before the exact maximum flow was found. Thus, the flow was smaller than the maximum flow. As we saw in Chapter 2, the weight of the minimum subgraph is found from the calculation: MXFLOW-R*$|E_G|$ where $|E_G|$ is the number of edges in the graph and MXFLO is the maximum flow. If we use a flow that is smaller than the maximum flow in this formula, we see that

the weight of this "minimum subgraph" would be smaller than that of the actual minimum subgraph. Thus by decreasing the space-time cost by the amount determined from a sub-maximum flow, the resulting space-time cost will be lower than the actual minimum space-time cost. For the R = 50 case we bounded the space-time cost between 3,014,925 and 3,289,095. The average of these two numbers estimates the actual space-time cost within $\pm$4%.

For the other two cases, the flow used is obtained from an initial flow. The initial flow is the flow from the source to a link node between each of the two link edges. This proportion is determined by the relative capacities of the terminal edges of the two nodes (other than the link node) connected to the two link edges. Using this flow we estimate space-time cost within 1.5% for R equals 500 and within .15% for R equals 5000.

In Figure 27, the solid line for DMIN connects the upper bounds on the minimum space-time cost. The vertical line below these points is connected to the lower bound. In Figure 28, the page fault curve labeled DMIN is computed from the upper bound of the space-time cost plotted in Figure 27.

In Figures 27 and 28, we see that for the smaller page size, there is again a substantial savings in space-time cost over the larger page size. We are assuming that despite the use of the smaller trace length for the 512 byte page experiment, we can compare the 4096 byte experiment with the 512 byte page experiment. This can be accomplished by multiplying the space-time cost from the 512 byte experiment by the ratio of the full trace length to the smaller trace length. For R equals

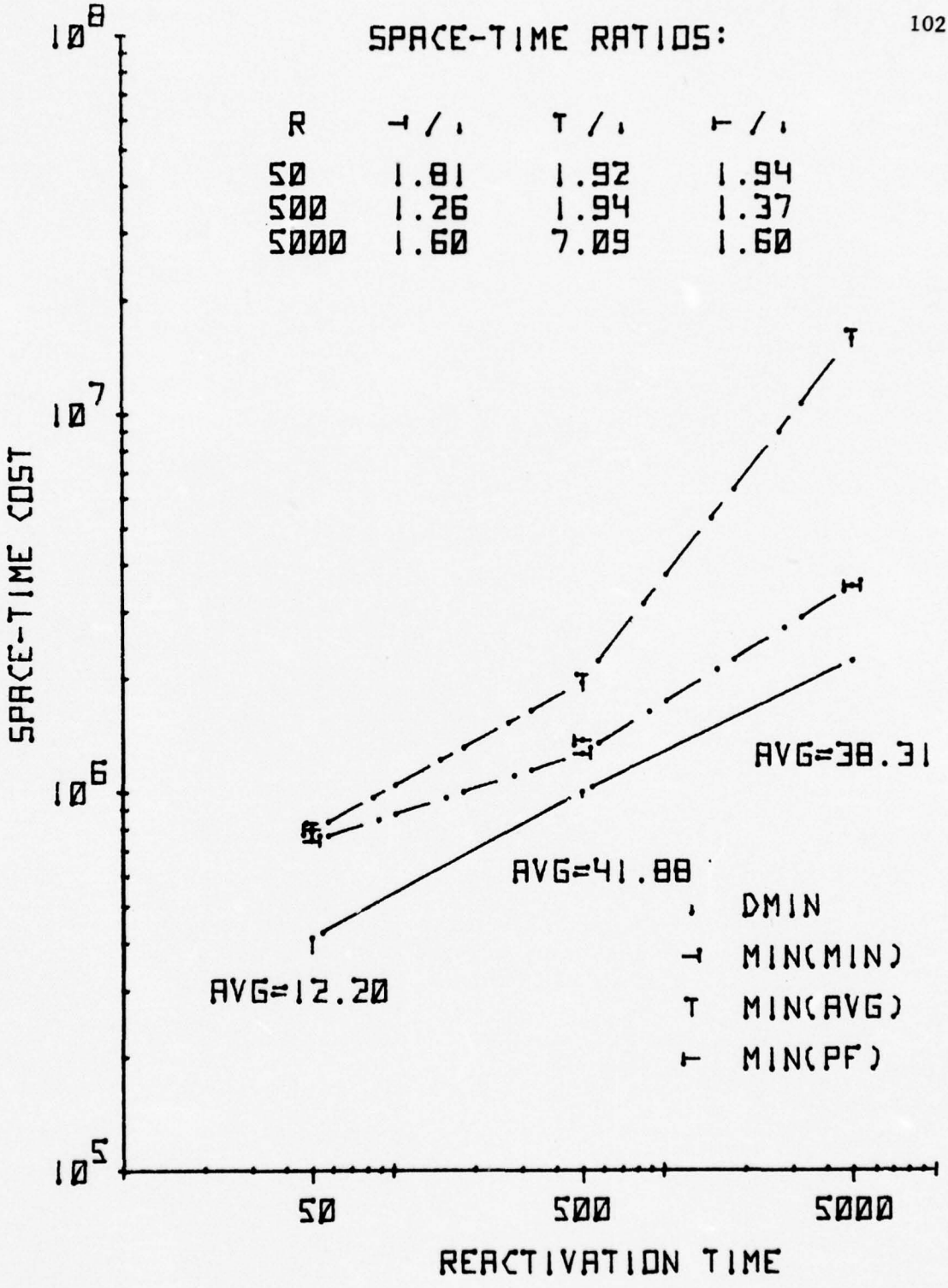


Figure 27. Space-time cost (normalized to a 4096 byte page) of MIN with DMIN for LIST with a 512 byte page.

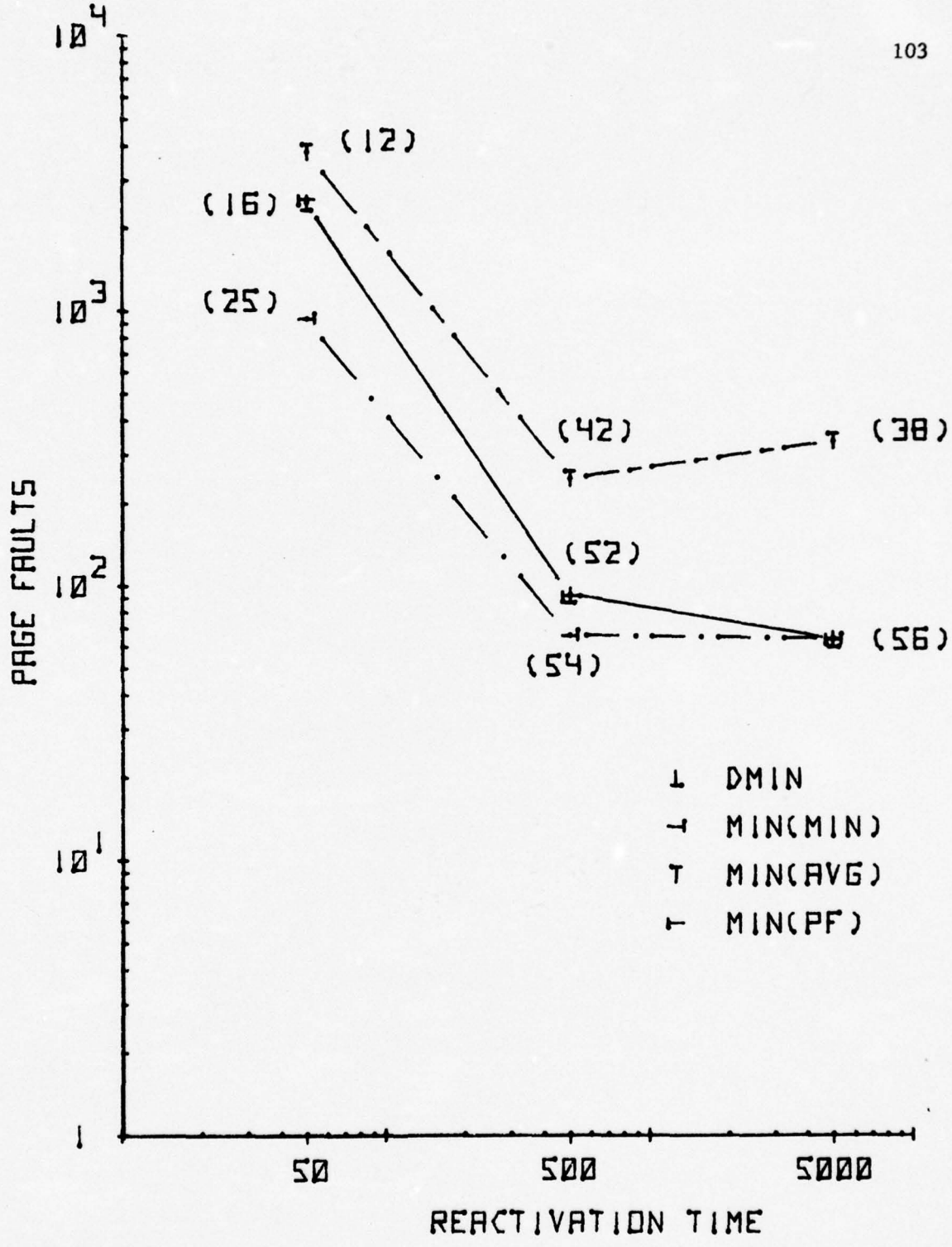


Figure 28. Page fault comparison of MIN with DMIN for LIST with a 512 byte page.

50, the space-time ratio for DMIN of LIST with a 4096 byte page to LIST with a 512 byte page is 2.86. For R equals 500, the space-time ratio is 1.84. For R equals 5000, the ratio is 1.05. The trend of decreasing space-time ratio with increasing R is explainable with the same conjecture that is used for explaining the GAUSS experiments.

Furthermore, the space-time cost of MIN(MIN) relative to DMIN is larger for the 512 byte page size. We conjecture that during execution with the smaller page size, there are more changes in the working set that minimizes space-time cost. Therefore, for the larger page size, there are relatively fewer page faults for DMIN to schedule. Thus suboptimal algorithms may perform closer to the optimum level of performance.

As for the LIST experiment with a 4096 byte page, the space-time ratio of MIN(MIN) to DMIN with a 512 byte page decreases as R changes from 50 to 500. We conjecture that this occurs for the same reason as in the 4096 byte page. Again, there is a large decrease in the number of page faults for DMIN as R changes from 50 to 500, i.e., from 2487 to 93. Also, MIN(MIN) is the allocation with the fewest number of page faults for R equals 500. As R increases from 50 to 500, the space-time cost from DMIN increases substantially. Since MIN(MIN) schedules the minimum number of faults for this case, there are no unnecessary faults which could increase MIN(MIN)'s space-time cost.

For R equals 5000, we see that for LIST with a 512 byte page, the MIN(MIN) to DMIN space-time ratio is greater than for the 4096 byte page. Our conjecture is that since there are 4 times as many page faults for the 512 byte page compared to the 4096 byte page, the increase in R causes a

larger increase for MIN(MIN) for the 512 byte page. As for the 4096 byte page, DMIN has the same number of faults as MIN(MIN). Thus, DMIN saves 50% of the space-time cost used for MIN(MIN) by removing pages after their last reference. Some pages make their last reference before all the pages have been referenced. Thus, there are fewer pages in the memory during the page faults caused by first references to a page for DMIN.

For this experiment, the MIN(AVG) allocations have substantially higher space-time costs than MIN(MIN), particularly for large R. We conjecture that this occurs because DMIN varies substantially from its average size of allocation.

4.2.3 Summary of DMIN versus MIN

We now summarize the space-time cost of DMIN and MIN. We have plotted the MIN(MIN) to DMIN space time ratio in Figure 29. In Figures 29 and 30, "G" represents the GAUSS program, "L" represents the LIST program; "4" represents the 4096 byte page and "5" represents the 512 byte page. We use these representations in the following discussion.

For reactivation time of 50, we conjecture that the order of the space-time ratios is dependent upon the relative variation in the working set sizes from AVG to DMIN. In Figures 23 and 25 we see that the LIST program's working set size varied much more with time than did the GAUSS program's working set. Also, for R equals 50, DMIN scheduled many more faults for the LIST program than for the GAUSS program. We conjecture that the smaller page size causes more time variation in the working set that minimizes space-time cost, than does the larger page. This is indicated by the large increase in page faults for the 512 byte page experiments compared to the 4096 byte page experiment.

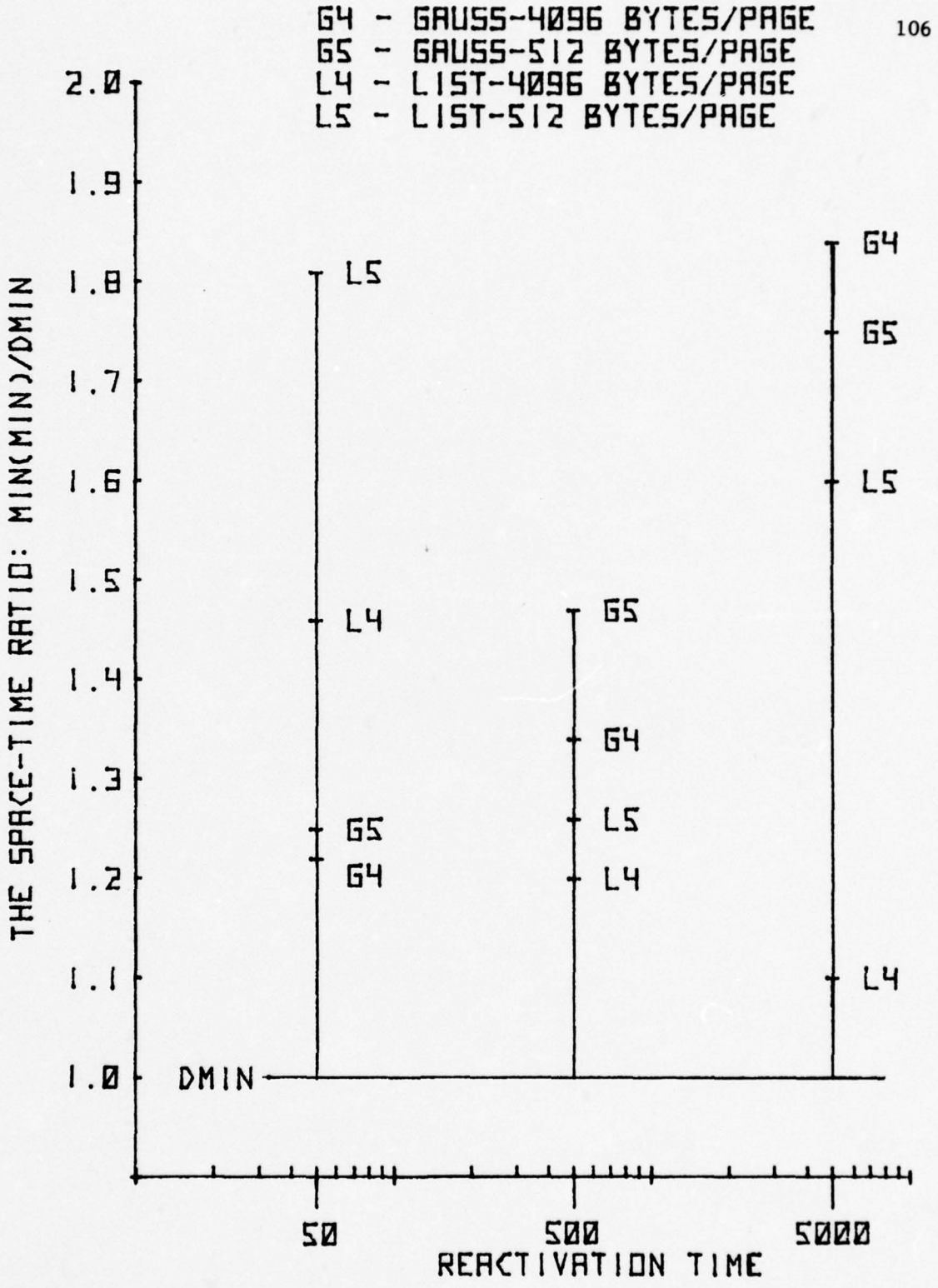


Figure 29. Summary of space-time cost ratios of MIN(MIN) to DMIN.

For reactivation times of 500 and 5000, we see a reversal in the ordering of the ratios with respect to the programs. We conjecture that this reversal occurs because for the larger reactivation times, the DMIN allocation cannot take advantage of the changes of the working set. This happens because it becomes more costly to vacate some intervals than to occupy these intervals as R increases. This affect is more pronounced in the LIST experiments since the working set that minimizes space-time cost varies more with time than for GAUSS. Thus, for the larger reactivation times, the space-time cost grows more rapidly for the LIST experiments.

In Figure 29, we have plotted the ratios for the best performance of the MIN algorithm. We feel that for this to be a fair comparison, the complexity of the memory controller for obtaining this level of performance for a heuristic static allocation would be close to the complexity of a heuristic dynamic allocation in the following sense. If a static allocation is to have the best performance, the memory controller should decide the size of allocation from program behavior. Consider Figure 30. We have plotted the space-time ratios of MIN(1/2) to DMIN, where MIN(1/2) is obtained by fixing the size of allocation at 1/2 of the pages used during execution. We see that all but 2 of the twelve cases use more than twice as much memory as DMIN. (These 2 cases did better because this allocation size happened to be close to their MIN(MIN) allocation sizes.) Our point is that if the size of static allocation is not determined carefully from program behavior, then the static algorithms will likely perform much worse than MIN(MIN).

We feel that based on the results shown in Figure 30, a dynamic allocation strategy is justified over a static allocation strategy. A

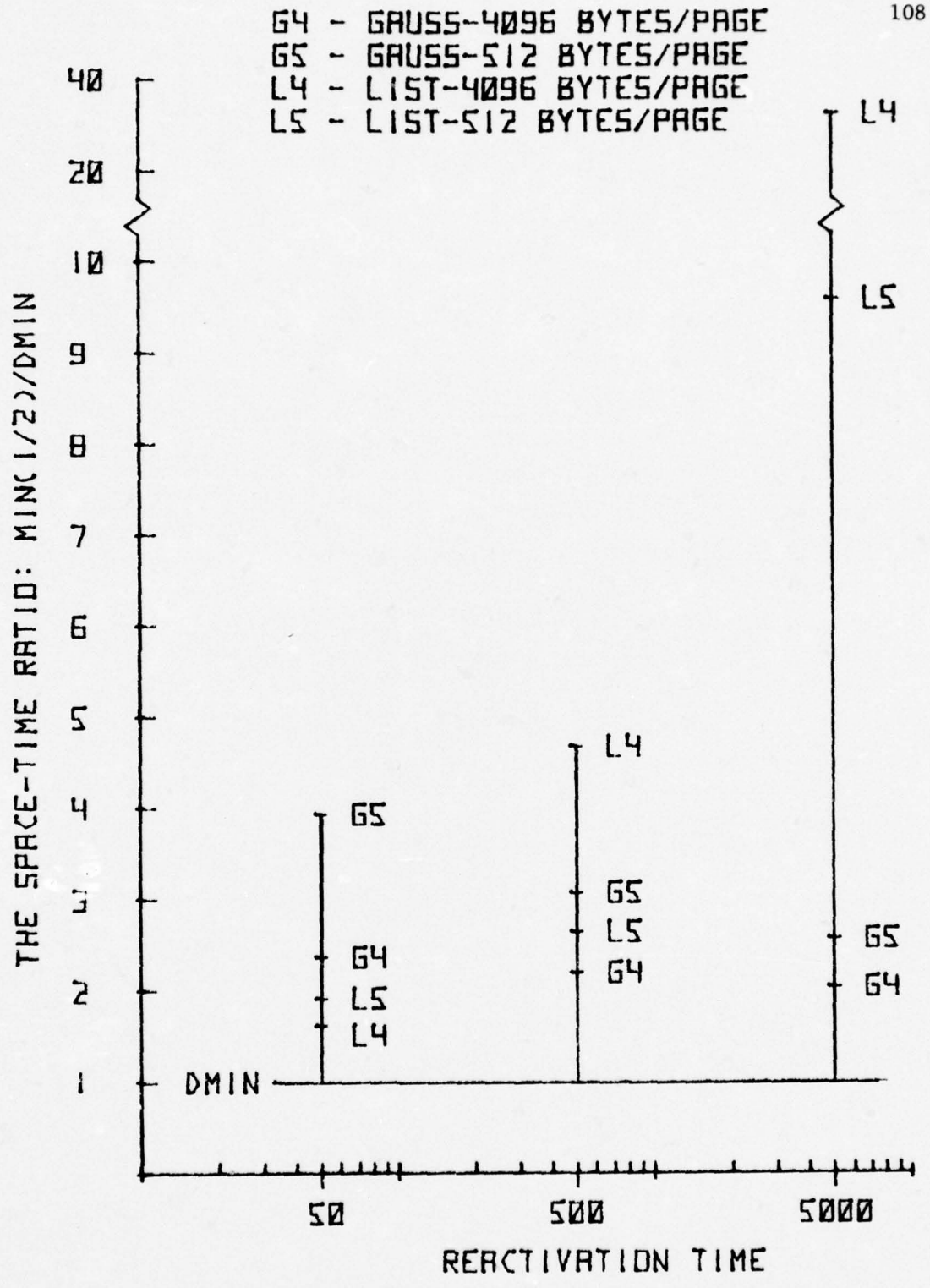


Figure 30. Summary of space-time ratios of MIN (1/2) to DMIN.

memory controller for a dynamic strategy would be more complex to implement compared to the simple strategy used for the static allocation in Figure 30. With the simple static strategy, the problem of programs requiring more primary memory space than exists is alleviated. However, the improvement for the dynamic strategy in these examples is 1.6 to 32.5 times the throughput. If the programs can be scheduled without running out of memory, then in any time interval we could expect to have twice as many memory references made for the dynamic strategy over the static strategy. If half of this potential can be realized, throughput would improve by 50%. Thus if one assumes that the complexity of the allocation algorithms is close (e.g., LRU vs PFF), the improved throughput for the dynamic strategy is achieved at a cost of a somewhat more complex memory controller and program swapping. This discussion assumes that the relative performance of heuristic static and dynamic allocation algorithms is comparable to the performance of the optimal allocation algorithms. Some more information regarding PFF is presented in Section 4.3.

In Figure 29, the ratios are closer to pure improvement for the dynamic strategy over the static strategy with the best possible allocation size. In order for the static strategy to achieve this level of performance, the memory controller would probably be as complex as for the dynamic strategy, i.e., both controllers would vary sizes of allocations for the start of each program's execution. Thus at the start of a program's execution, the static strategy memory controller could require more primary memory than is available. However, this seems to be more of a problem for a dynamic memory controller since running out of memory could

occur anytime within a program's execution. However, this exceeding of the primary memory would be most likely to occur for programs with large variations in their sizes of working sets. From the above experiments these variations in working set size would be exploited for small reactivation time and small page size. However, from Figure 29, we see that these are the conditions which allow the improvement in space-time ratio for LIST, a program with large variations in its working set size.

From these results and with these assumptions, we conjecture that a conservative estimate of improved performance achieved by a dynamic strategy compared to a static strategy would be in the range of 10% to 40% better memory utilization. The 40% figure seems to be more likely achieved for a program like LIST as reactivation time decreases, page size decreases, and the size of the working set varies significantly, but slowly compared to reactivation time. Thus a memory architecture with fast secondary memory and a small page size would have the biggest improvement in performance by using a dynamic allocation strategy for a program like LIST.

For the GAUSS program, the number of page faults scheduled by DMIN is close to the number scheduled by the MIN(MIN) allocation. For the 4096 byte page DMIN scheduled fewer faults for R equals 50 and 500. For the 512 page, DMIN scheduled no more than 30 (38%) more faults than MIN(MIN). For the LIST program DMIN scheduled as many as 1546 more faults then the MIN(MIN) allocation. The biggest difference in page faults occurred for the case R equals 50 and for the 512 byte page. However, it is reasonable to assume that for the computer architecture configured with a small reactivation time and a small page size, a large number of page faults could be handled.

We do not consider the high number of faults as a reason not to use dynamic allocation strategy. One reason is that a heuristic dynamic algorithm in a computer would be more conservative in causing potential page faults if the memory architecture is designed only to handle a low fault rate, i.e., if R increases rapidly as the fault rate goes up. Also, if the secondary memory bandwidth is saturated, the memory controller could attempt to limit page faults by program scheduling and/or by dynamically varying the R parameter of the allocation algorithm or equivalent heuristic parameters. Finally, DMIN scheduled the most page faults for the program with the most variation in size of working set. This type of program is likely to perform most poorly for a static allocation relative to a dynamic allocation. If the static allocation is too small, the program will thrash. If the allocation is too large, the program will waste memory.

For a program like GAUSS, we would expect a dynamic allocation to achieve better memory utilization compared to a static allocation as reactivation time increases. For the GAUSS program, the smaller page size produced better results for reactivation times of 50 and 500 and worse results for R equals 5000. Thus, a memory architecture with a large reactivation time and a large page size would have the biggest improvement in performance by using a dynamic allocation strategy for a program like GAUSS. These results contrast with those for a dynamic allocation for the LIST program. A dynamic allocation has the best improvement over static

allocation at R equals 50 with 512 byte page for LIST. Thus, we conclude that under a general job load, a dynamic allocation could be beneficial for a broad range of memory architectures.

On a global level, memory overflow can occur under a dynamic allocation strategy when the active jobs collectively demand more space than is available. In a time-sharing environment with a heavy job load, memory overflow cannot be avoided at times. In order to allow active jobs to run efficiently, jobs should be swapped out if they cannot all fit in the memory. New or suspended jobs should be activated if memory is under-utilized. In a batch environment, the number of active jobs may be controlled without job swapping. For example, while jobs are running and being completed, a decision can be made about whether to add new jobs. This decision is dependent upon whether sufficient space remains in the memory for another job to run to completion. New jobs can be started whenever sufficient space is available.

Furthermore, for both time-sharing and batch systems with a dynamic allocation strategy, the parameters of the allocation algorithm can be adjusted dynamically in an attempt to stop the memory from overflowing. By varying the allocation algorithm parameters, the memory controller is deciding whether it is better for system performance to swap jobs out or reduce the amount of space which resident jobs occupy. In contrast, a static allocation is an open loop process. Once the size of allocation has been guessed at, the job executes with this allocation regardless of whether there is too much or too little memory space for the job to run efficiently.

4.3 Experimental Results for DMIN versus PFF

In this section, we evaluate the Page Fault Frequency (PFF) algorithm by comparison with the optimum DMIN algorithm for the three chosen reactivation times, two page sizes, and for both programs. The PFF algorithm is a function of P [6], the page fault frequency parameter. As described above we use T, the inverse of P. For each program and each page size, we increase T until the number of faults is minimum, i.e., the only faults that occur are caused by the first reference to a page. We decrease T until the space-time cost begins to increase.

In Figure 31 we compare the PFF algorithm with DMIN for the GAUSS program with a 4096 byte page. In this figure we have listed the T parameter and the number of page faults which occur with each value of T in a table associated with R = 50. The values of T listed are the ones used for the experiments. In the other T tables, we have omitted the number of faults caused by each value of T since the number of faults is the same for the same value of T. In the T tables, the ordering in each table is such that in going from top to bottom, the space-time cost increases for the experiment as we go down the table. Thus, the value of T in the top row is the T that caused the least space-time cost, and the value of T in the bottom row caused the most space-time cost. Each T table is related to a fixed value of R. The T table on the left is for R equals 50; the T table on the right is for R equals 5000. The number of page faults scheduled by DMIN for a given value of R is placed in parentheses near the space-time cost plot resulting from the DMIN allocation for that given R. In the figure, PFF(MIN) represents the minimum value of space-time cost resulting from the PFF experiments performed for that particular value of R.

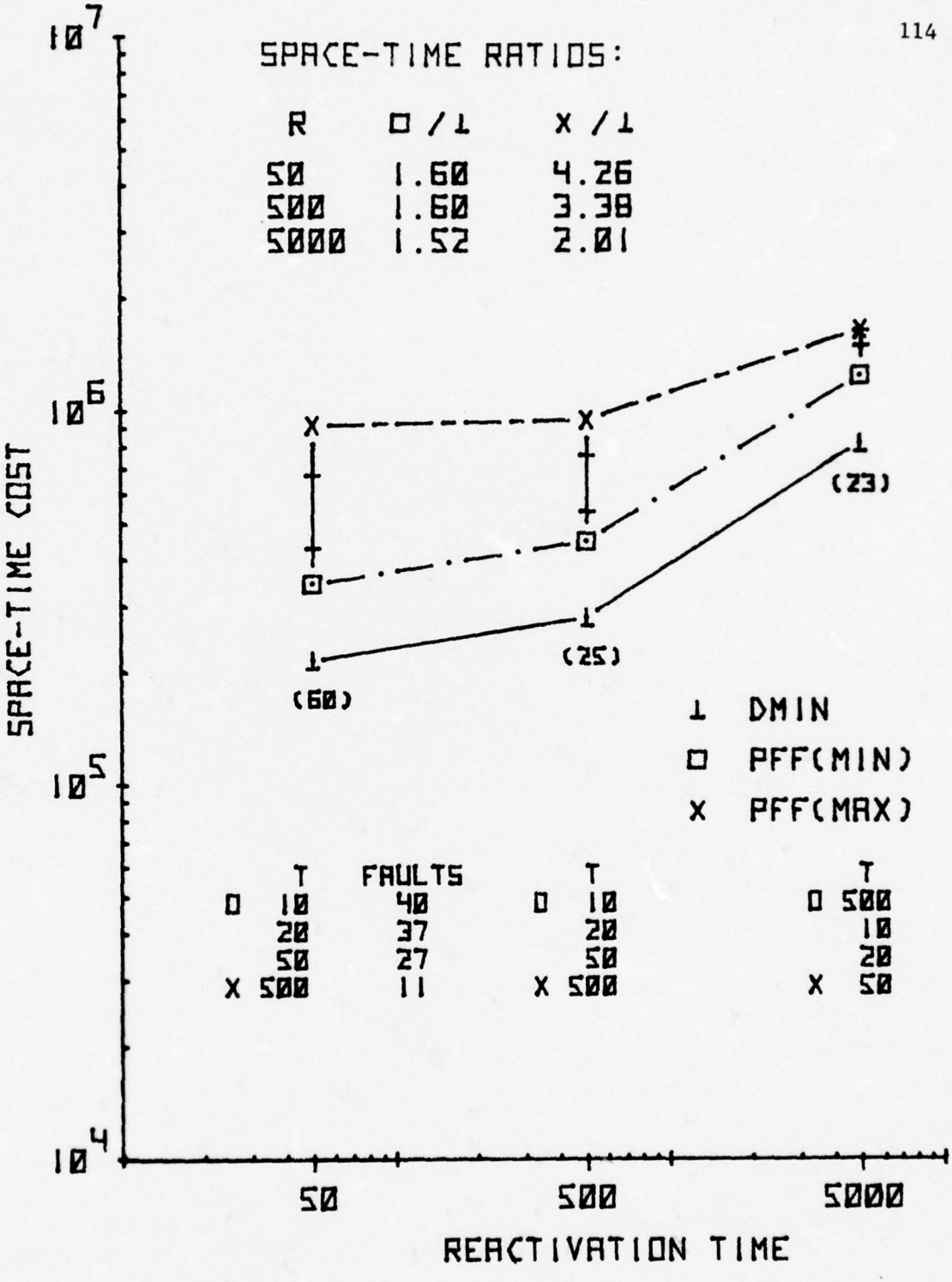


Figure 31. Comparison of PFF with DMIN for GAUSS with a 4096 byte page.

Similarily, PFF(MAX) represents the maximum value resulting from the experiments. This discussion also applies to Figures 32, 33, and 34.

In Figure 31 we see that the performance of PFF is substantially poorer than the performance of DMIN. At best, PFF uses 52% more memory space-time than does DMIN. Also, the performance of PFF is sensitive to T. For R equals 50, the space-time cost for T = 50 is nearly twice as large as for T = 10. For R equals 500, the space-time cost for T equals 50 is about 1.7 times greater than for T equals 10. The number of page faults scheduled by DMIN is similar to the number of faults for PFF(MIN).

In Figure 32 we compare PFF with DMIN for the GAUSS program with a 512 byte page. In comparing this figure with Figure 31, we see that the performance of PFF has improved for R equals 50, but the performance has decreased for R equals 5000.

For both Figures 31 and 32, the performance is most sensitive to T for R equals 50 and markedly less sensitive at R = 5000. At R equals 50, the space-time cost for T equals 500 is 2.59 times greater than the cost for T equals 20. The number of page faults scheduled by DMIN is generally within the range of the number of page faults caused by the PFF(MIN) allocations.

In Figure 33 we compare PFF with DMIN for the LIST program with a 4096 byte page. In this figure we see that the relative performance of PFF(MIN) improves with increasing R, while PFF(MAX) deteriorates at large R. The space-time ratios of PFF(MIN) to DMIN decrease with increasing R. However, the sensitivity of the T parameter is greatest for R equals 5000, in contrast to the results for GAUSS. For R equals 5000, the space-time cost for T equals 100 is a factor of 3.19 greater than the cost for T equals 500. We conjecture that the improvement in performance as R

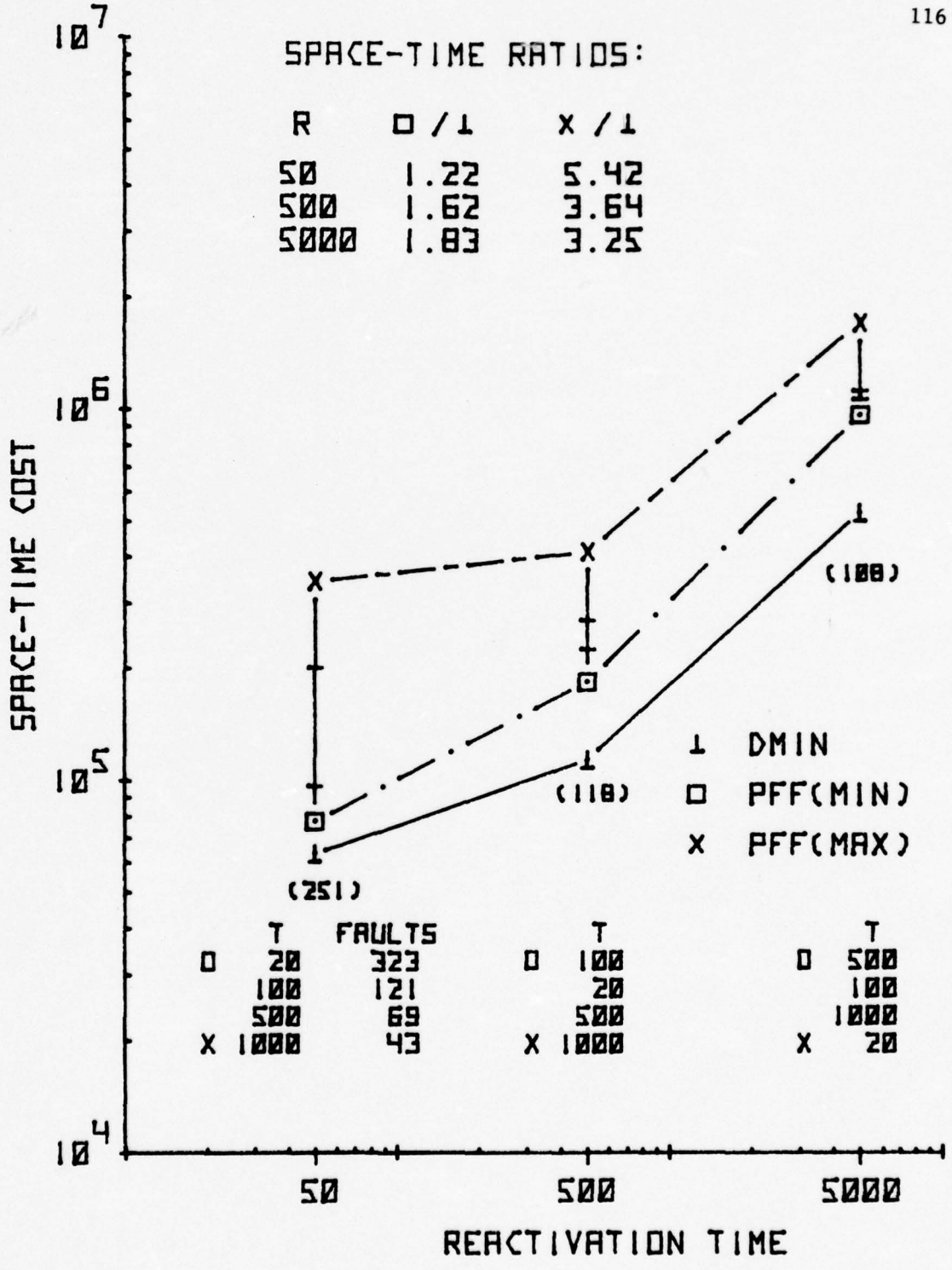


Figure 32. Comparison of PFF with DMIN for GAUSS with a 512 byte page (space-time cost normalized to a 4096 byte page).

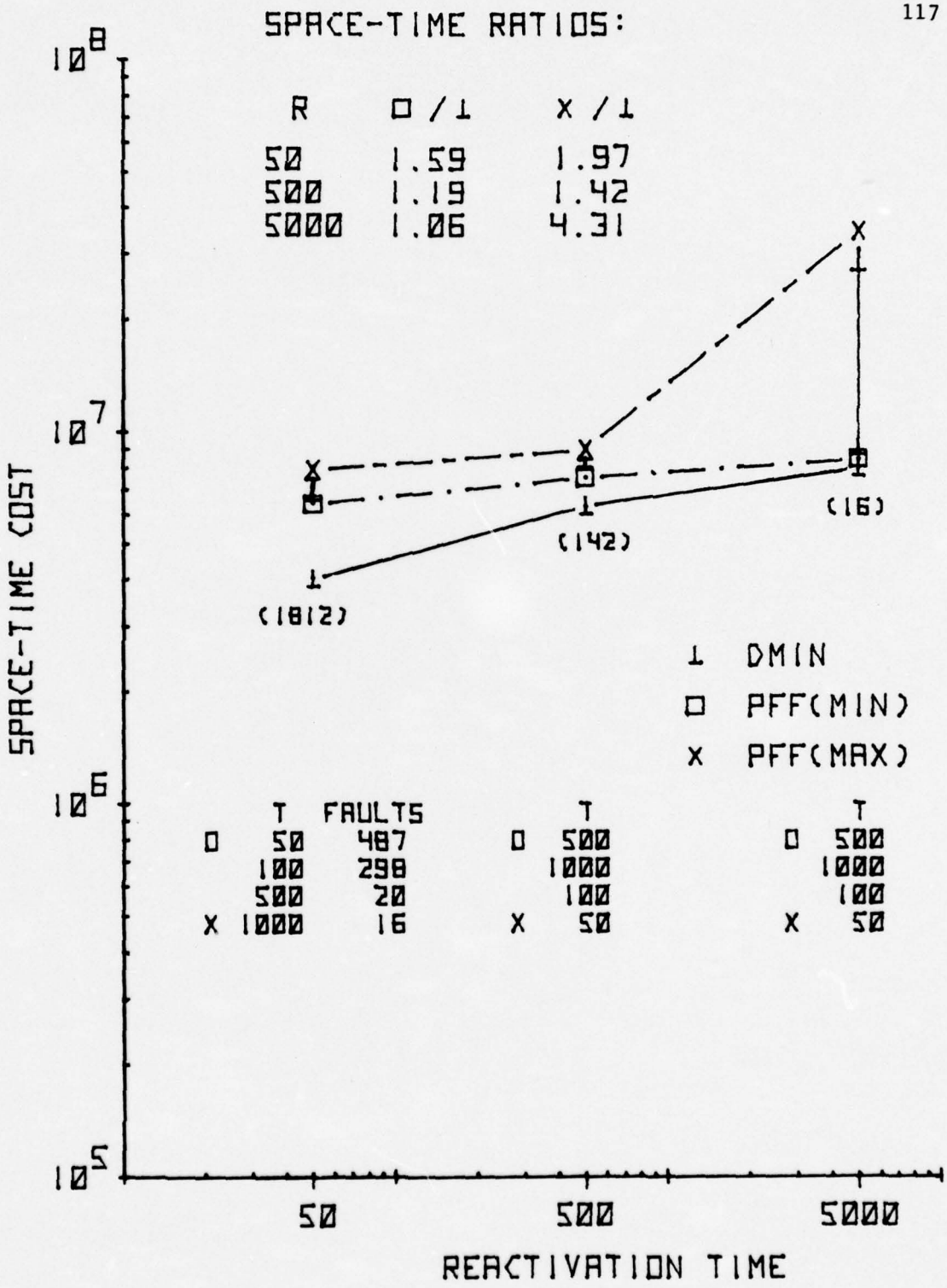


Figure 33. Comparison of PFF with DMIN for LIST with a 4096 byte page.

increases in the PFF(MIN) allocation is due to program behavior. As we saw in the previous discussion the LIST program's working set changes substantially during the execution of the program. However, in the optimal allocation for the larger values of reactivation time, we conjecture that it becomes more costly to remove some pages during some periods when they are not in the working set. This is indicated by the large decrease in the number of page faults as R changes from 50 to 500. When DMIN schedules few page faults, PFF(MIN) comes closest to matching DMIN in space-time cost.

In Figure 34 we compare PFF with DMIN for the LIST program with a 512 byte page. We see that the PFF performance relative to DMIN is qualitatively similar to the performance for the LIST program with a 4096 byte page. However, the numerical results have a far wider spread for the smaller page size. The performance of PFF(MIN) relative to DMIN improves as R increases. However, sensitivity to T is greatest at R equals 5000. For R equals 5000, the space-time cost for T equals 1000 is 2.71 times greater than the cost for T equals 500.

In looking over these results we see certain trends in the comparisons between the PFF algorithm and DMIN. For small reactivation time, the PFF achieved a space-time cost of 1.22 to 1.83 that of DMIN, with a carefully chosen value of T. For a poorer choice of R, relative space-time varied from 1.42 to 12.07. The closeness of PFF to DMIN and the sensitivity of PFF to T as a function of R varied for the two programs and the page sizes used. The performance of PFF may erode substantially for poorly chosen T. For the LIST program, the PFF algorithm has better relative performance for larger values of reactivation time. However, for the above experiments with the LIST program, the space-time cost was very sensitive to variation in T.

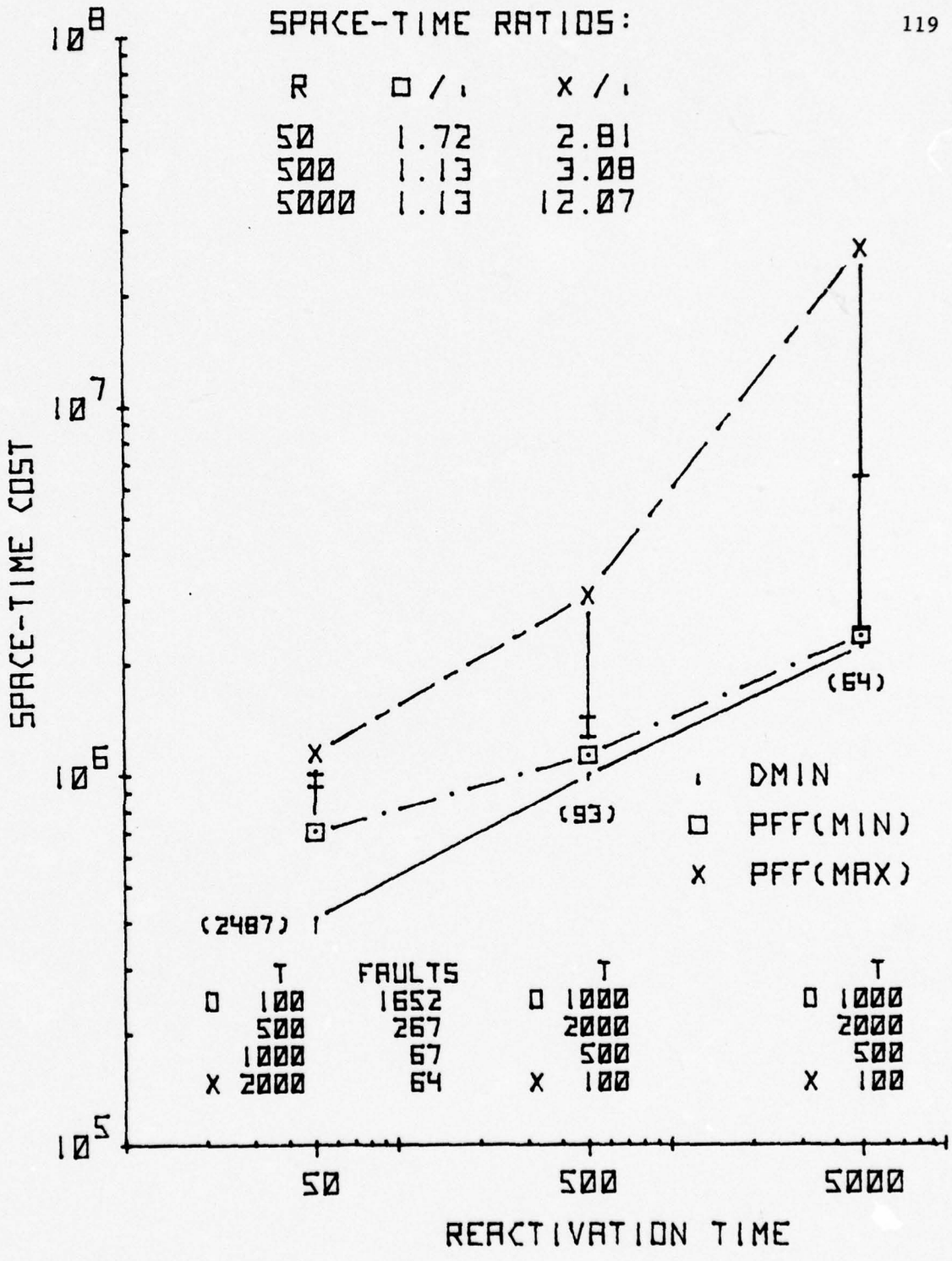


Figure 34. Comparison of PFF with DMIN for LIST with a 512 byte page (space-time cost normalized to a 4096 byte page).

Thus, we conclude that it is unlikely that the PFF(MIN) allocations would be achievable for the PFF algorithm in practice.

We do not feel that the PFF algorithm is the best achievable heuristic. The space-time cost performance relative to DMIN was sporadic. PFF performed very well with respect to space-time cost for the LIST program with both page sizes and reactivation times of 500 and 5000. However in the other cases, this performance was not maintained. Another problem with the PFF algorithm is its sensitivity to the P parameter. The P parameter must be carefully chosen. Otherwise, space-time cost could increase substantially over the PFF(MIN) performance.

4.4 The τ Distributions for Optimal Allocations

In this section, we present the distributions of the τ's for two of the LIST experiments. We consider the LIST program with a 4096 byte page for reactivation times of 50 and 500. We discuss the distribution of the lengths of the τ's which are vacated for these two experiments.

In Figure 35 we illustrate the τ distribution for the LIST program with a 4096 byte page for R equals 50. The height of each bar is the number of nonreference intervals with a τ in the corresponding range. At the top of each bar, we place the per cent of the intervals in that bar which are occupied in the optimal allocation. In the figure, the part of the bar which has an X in it is proportional to the number of intervals which are occupied. The part of the bar without the X is proportional to the number of intervals which are vacated in the optimal allocation. These same conventions are used in Figure 36.

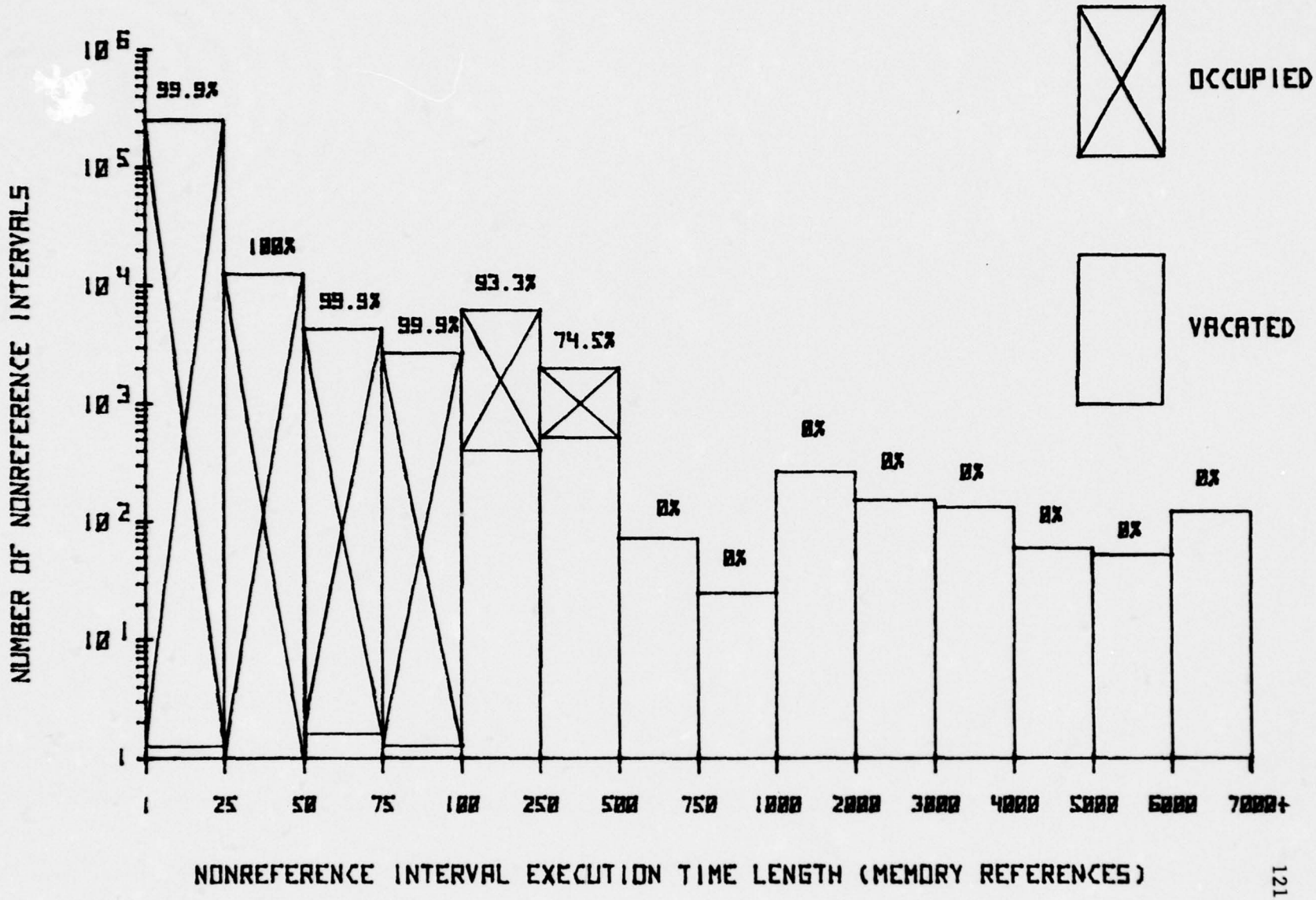


Figure 35. The τ distribution for LIST with a 4096 byte page and reactivation time of 50.

In Figure 35, we see that all intervals with τ's greater than or equal to 500 are vacated. There are 880 intervals with τ's greater than or equal to 500. This is just under half of the 1796 vacated intervals from the DMIN allocation. Just 4 of the intervals with τ's less than 100 are vacated. Thus, Denning's Working Set Algorithm [4] with a window size of 500 memory reference would have vacated 900 intervals that save space-time cost for the DMIN allocation. However, the Working Set algorithm would not have saved as much cost as DMIN since DMIN vacates the intervals at the start of the nonreference interval. Thus some of the intervals that would be vacated by the Working Set algorithm would probably increase space-time cost. In the range of τ's from 100 to 499 memory references, we see that 912 intervals are vacated. This is just over half of all the intervals vacated. However, if the working set parameter is set to 100 memory references, the Working Set algorithm would deallocate these 912 intervals along with 7241 other intervals which DMIN did not vacate for τ's in the range of 100 to 499 memory references. Thus, this Working Set algorithm would be expected to perform poorly compared to DMIN if these 912 faults saved a substantial amount of space time cost.

In Figure 36, we illustrate the τ distribution for the LIST program with a 4096 byte page, for R equals 500. We see that the number of vacated intervals in Figure 36 are less than 10% of the number vacated in Figure 35. Also, all the vacated intervals in Figure 36 have τ's which are greater than 3000, 6 times the reactivation time used to generate this data. In Figure 35, one interval has a τ smaller than 50, the reactivation time used to generate the data in this figure. In Figure 36,

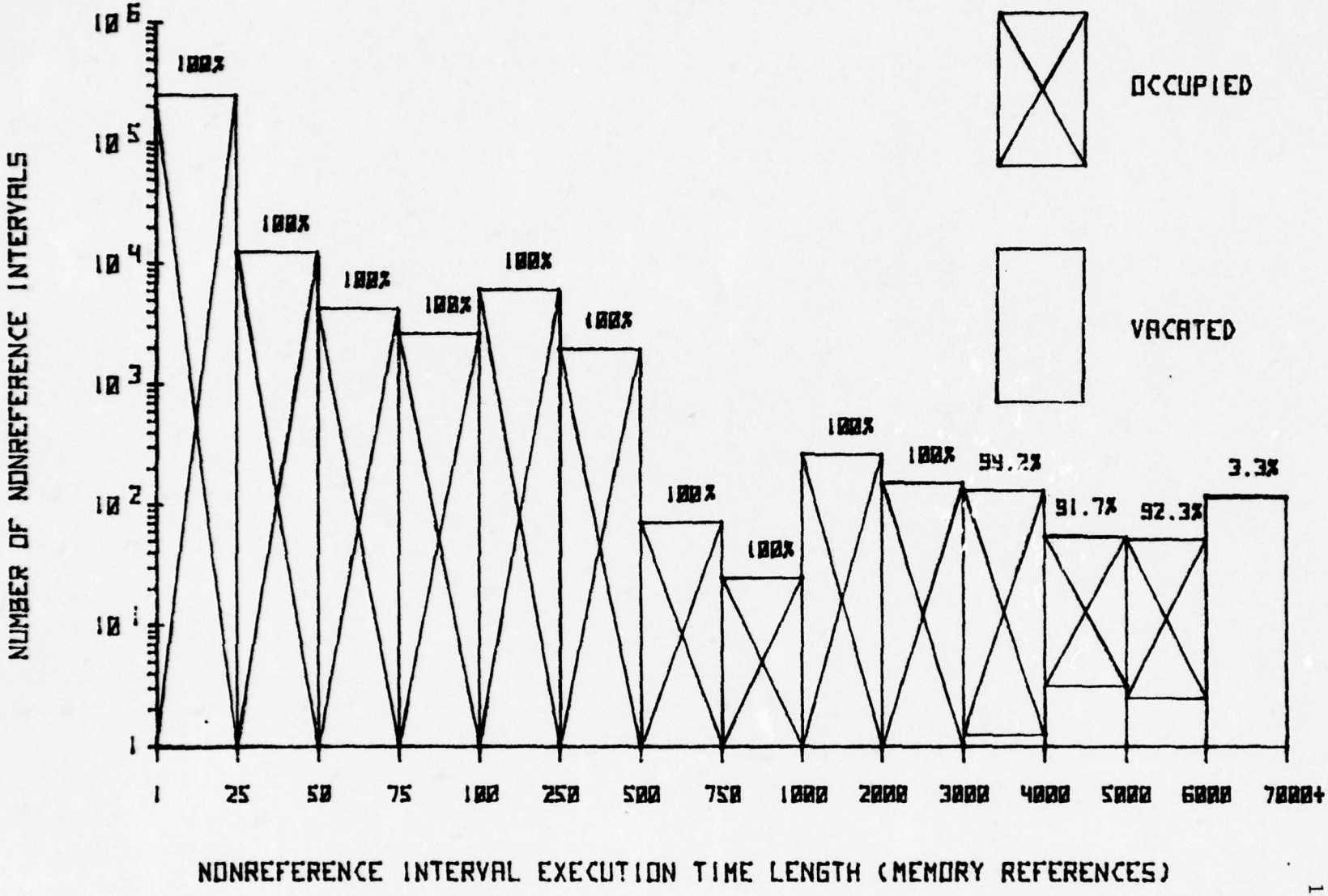


Figure 36. The τ distribution for LIST with a 4096 byte page and reactivation time of 500.

less than 5% of the vacated intervals have τ's less than ten times the reactivation time. In Figure 35, more than 50% of the vacated intervals have τ's less than ten times the reactivation time.

The reason that intervals vacated by DMIN with R equals 50 tend to have smaller τ:R ratios compared to the τ:R for R equals 500 is that for the smaller R, intervals are vacated together in clusters, i.e., intervals tend to be included in negative weight subgraphs even when τ is not large. For R equals 500, this clustering of vacated intervals does not occur for τ near 500. Thus, for R equals 500, minimum weight subgraphs do not contain intervals whose τ is near 500. Thus since these minimum subgraphs no longer occur for R equals 500, we can infer that by increasing the reactivation time from 50 to 500 the node weights increase more rapidly than the edge weights.

For the Figure 36 data, Denning's Working Set algorithm would perform closely to DMIN for a working set parameter of 7000 memory references. With this parameter value, the Working Set algorithm would schedule 106 of the 116 vacated intervals that DMIN did. Only four intervals would be erroneously vacated compared to DMIN.

For a 512 byte page, we expect that the Working Set algorithm would perform more poorly than it did for the 4096 byte page. With a 512 byte page, we would expect more clustering of vacated intervals for the DMIN allocation, especially for small reactivation times. With the clustering effect, the τ threshold which separates vacated intervals from occupied intervals tends to become more ambiguous. The threshold is ambiguous when over several adjacent ranges of τ values, the intervals whose τ's fall within each range are separated into a substantial number

of occupied intervals and a substantial number of vacated intervals. The threshold becomes more ambiguous with the 512 byte page because within a given range of τ's, it seems likely that some of the associated intervals will cluster and others will not. Thus if the clustering effect occurs over a wide range of τ's, the Working Set will have difficulty in vacating the intervals that DMIN vacates unless it vacates a substantial number of intervals occupied by DMIN. Another problem with the Working Set algorithm is choosing a "good" value for its parameter.

4.5 Experimental Results for DMIN versus DWS

In this section we evaluate the Damped Working Set (DWS) algorithm by comparison with the optimum DMIN algorithm. We compare DWS with DMIN for the LIST program for the three chosen reactivation times and the two page sizes. The DWS algorithm is a function of two parameters: W, the working set parameter, and MULT, the multiplicative parameter [5]. For the 4096 byte page experiments, we choose W's from the τ distribution data from the DMIN allocations illustrated in Figures 35 and 36. For these experiments, we used the MULT parameter values of 1, .5, and .25. We first used a MULT value of 1, which then makes DWS equivalent to Denning's Working Set algorithm. If we thought that the data justified increased page faults, we used MULT values of .5 and .25. For the 512 byte page experiments, we were forced to guess at appropriate values of W. Unfortunately previous experiments did not reliably indicate the best choices for W. Thus, we ran a large number (12) of these experiments. For these experiments, the data justified only the use of MULT parameter value of 1.

In Figure 37, we illustrate the performance of DWS relative to DMIN for the LIST program with a 4096 byte page. In this figure we plotted DWS for various values of W and for a MULT parameter value of 1. We did not plot the DWS values for other values of MULT because the space-time cost for these values was at most 5 from the MULT equals 1 case. In this figure we use DWS(MIN) to define the minimum space-time cost achieved by DWS for a particular value of reactivation time. The DWS(MIN) values are obtained with three values of W. These three values are listed in the table below the R equals 50 points. We listed the faults associated with each value of W. The number of faults associated with a particular value of W does not change as R varies.

The other plotted points from the DWS algorithm are labeled DWS(MAX). The W value associated with a DWS(MAX) value for a particular R is chosen from among the three values listed in the table below the R equals 50 points. The W value used from among these three values is the value which has maximum space-time cost. For each reactivation time, we list the value of W associated with DWS(MIN) and DWS(MAX) in a table below that value of reactivation time. The number of page faults scheduled by DMIN for a particular value of R is placed in parentheses near the space-time cost plot resulting from the DMIN allocation for that given R.

For Figure 37, we chose the W parameters partially from the τ distribution of intervals vacated by DMIN as shown in Figures 35 and 36. We chose a W value of 500 memory references because for R equals 50, all intervals with a τ greater than 500 are vacated in the optimal allocation. We chose a W value of 7000 memory references because for R equals 500, all intervals with a τ greater than 7000 are vacated in the optimal allocation.

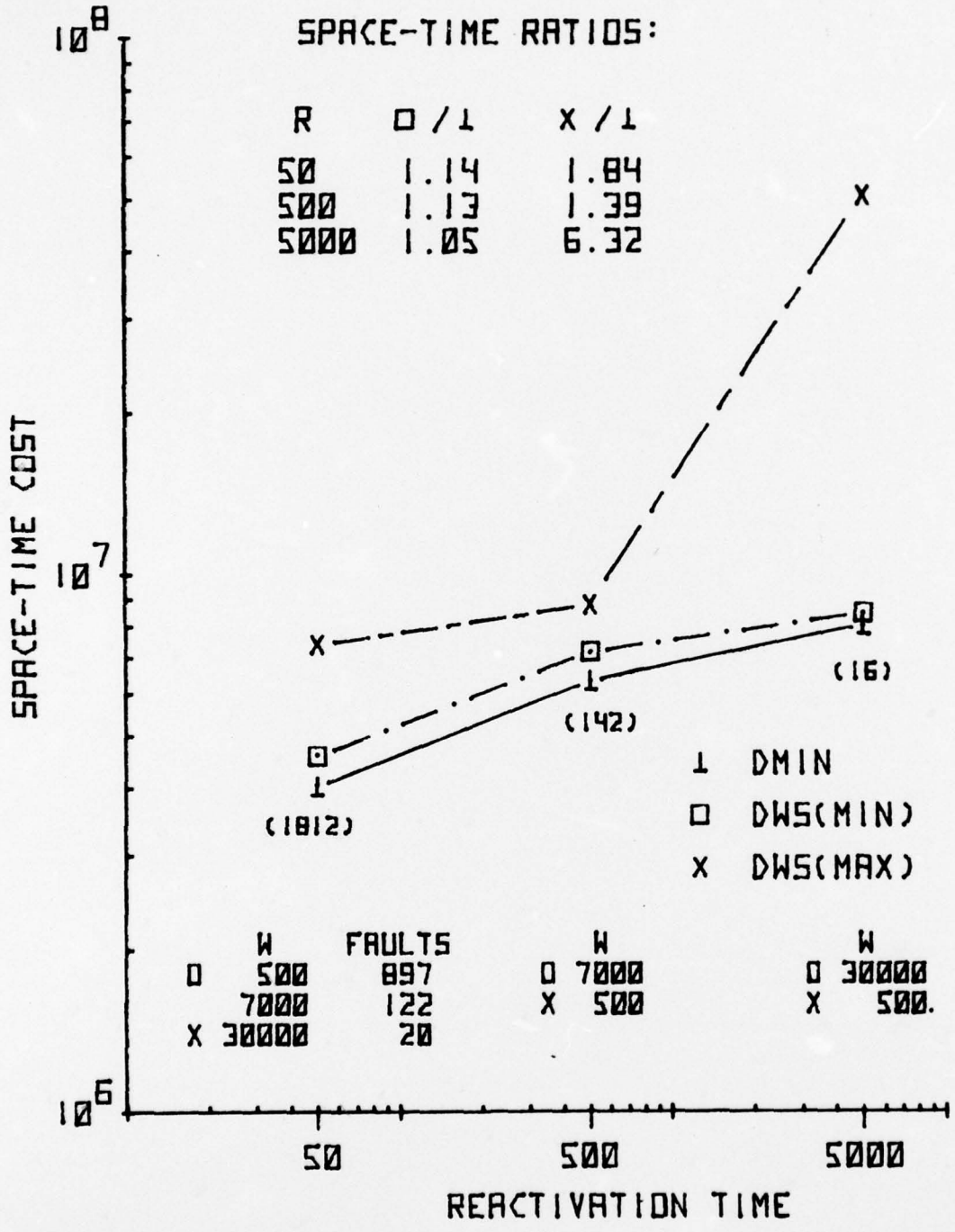


Figure 37. Comparison of DWS with DMIN for LIST with a 4096 byte page.

The other W value is 30,000 memory references. The τ distribution of vacated intervals in the optimal allocation offered no help for this choice of W. For R equals 5000, the DMIN allocation only deallocated pages after their last reference. We chose this remaining W value by increasing W until the DWS algorithm scheduled about the same number of faults as did the DMIN allocation. This occurred for W equals 30,000 memory references.

In Figure 37, we see that the DWS(MIN) allocation is very close to the optimal allocation for the LIST program with the 4096 byte page. For reactivation times of 50 and 5000, DWS(MIN) out-performed both the MIN and PFF algorithms for LIST with a 4096 byte page with respect to space-time cost. For R equals 500, the DWS(MIN) allocation matched the PFF(MIN) allocation. From the space-time ratios, we see that the DWS(MIN) allocation is no more than 14% of the optimal space-time cost. For R equals 50, we see that the DWS(MIN) allocation shedules 897 page faults. This number of faults is about half the number scheduled in the optimal allocation. For the other reactivation times, the number of faults from DWS(MIN) is close to the number of faults from DMIN.

In Table 3 we list the results of the DWS experiments for the LIST program with a 4096 byte page. In the W column, we list the values of W used in Figure 37. For W equals 500 and 7000, we used MULT values of 1, .5, and .25. For each value of W and MULT we list the corresponding number of faults scheduled by the DWS algorithm. Also, for each value of W, MULT, and R, we list the corresponding space-time ratio of DWS divided by DMIN. For comparison, we list both the space-time cost and page faults associated with the DMIN allocation for each value of reactivation time.

Table 3

Experimental Results of the DWS Algorithm for LIST with a 4096 Byte Page vs DMIN

W	MULT	Faults	SPACE-TIME RATIO: DWS/DMIN R: 50	500	5000
500	1	897	1.14	1.39	6.32
500	.5	1051	1.14	1.50	7.25
500	.25	1629	1.18	1.87	10.26
7000	1	122	1.78	1.13	1.74
7000	.5	148	1.60	1.15	1.96
7000	.25	194	1.57	1.18	2.34
30000	1	20	1.84	1.19	1.05

DMIN ALLOCATION	REACTIVATION TIME 50	500	5000
SPACE-TIME COST	4045610	6348253	8051180
FAULTS	1812	142	16

We see that varying the MULT parameter from a value of 1 substantially increases the number of page faults. For W equals 500, the number of faults for MULT equals .25 is nearly double the number of faults for MULT equals 1. For W equals 7000, the number of faults for MULT equals .25 is more than 50% greater than the number of faults for MULT equals 1.

We see that the best space-time performance occurred for MULT equals 1 in all but one case. For W equals 7000 and R equals 50, the space-time ratio for mult equals .25 is somewhat smaller than for MULT equals 1. Based on these observations we conclude that the DWS algorithm performs best when MULT has a value of 1, i.e., the Damped Working Set algorithm has the best performance when it is equivalent to Denning's Working Set algorithm.

From Table 3, we see that Denning's Working Set algorithm is rather insensitive to variation in W with respect to space-time cost. For R equals 50, we see that the space-time cost increases by a factor of 70% as W varies from 500 to 30,000 memory references. For R equals 5000, we do see an increase of more than 500% as W changes from 500 to 30,000. However, this is not a fair sensitivity measure since it is unreasonable to deallocate a page that has not been referenced for 500 memory references if it takes 5000 memory references to reactivate a program after a page fault. For R equals 5000, we see that there is an increase in space-time cost of less than 70% as W varies from 7000 to 30,000. Thus, we conclude that the space-time cost in these experiments with Denning's algorithm is not very sensitive to variations in the W parameter.

Based on Figure 37 and Table 3, our conclusion is that Denning's Working Set algorithm is an excellent heuristic algorthm for the LIST program with a 4096 byte page. The space-time cost from Denning's algorithm is consistently close to the optimum. The space-time cost performance is rather insensitive to changes in W, the working set parameter. The number of page faults compares very favorably with the DMIN allocations.

In Figure 38 we compare Denning's algorithm with DMIN for the LIST program with a 512 byte page. This figure has the same format as Figure 37. However, we show only two values of W since these two values produce the DWS(MIN) allocations for the three reactivation times used. We did not use a MULT value other than 1 since the data seems to indicate that the other values of MULT would not improve the space-time cost performance. In Figure 38 we used a vertical line for the DMIN points. This is done because we are only able to bound the performance of the DMIN allocation. The optimal space-time cost is within the span of the vertical line. For the space-time ratios in this figure, we used the upper bound of the space-time cost from the DMIN experiments.

For Figure 38, we did not have the τ distribution from the DMIN allocation because we only bounded the space-time cost within a range. The W values used in this figure were obtained by performing 12 experiments.

For Figure 38, we see a great difference in the performance compared to Figure 37. For LIST with a 512 byte page we see that DWS(MIN) is at best more than 2.5 times the upper bound of the DMIN allocation. In Section 4.4, we conjectured that the performance for the 512 byte page

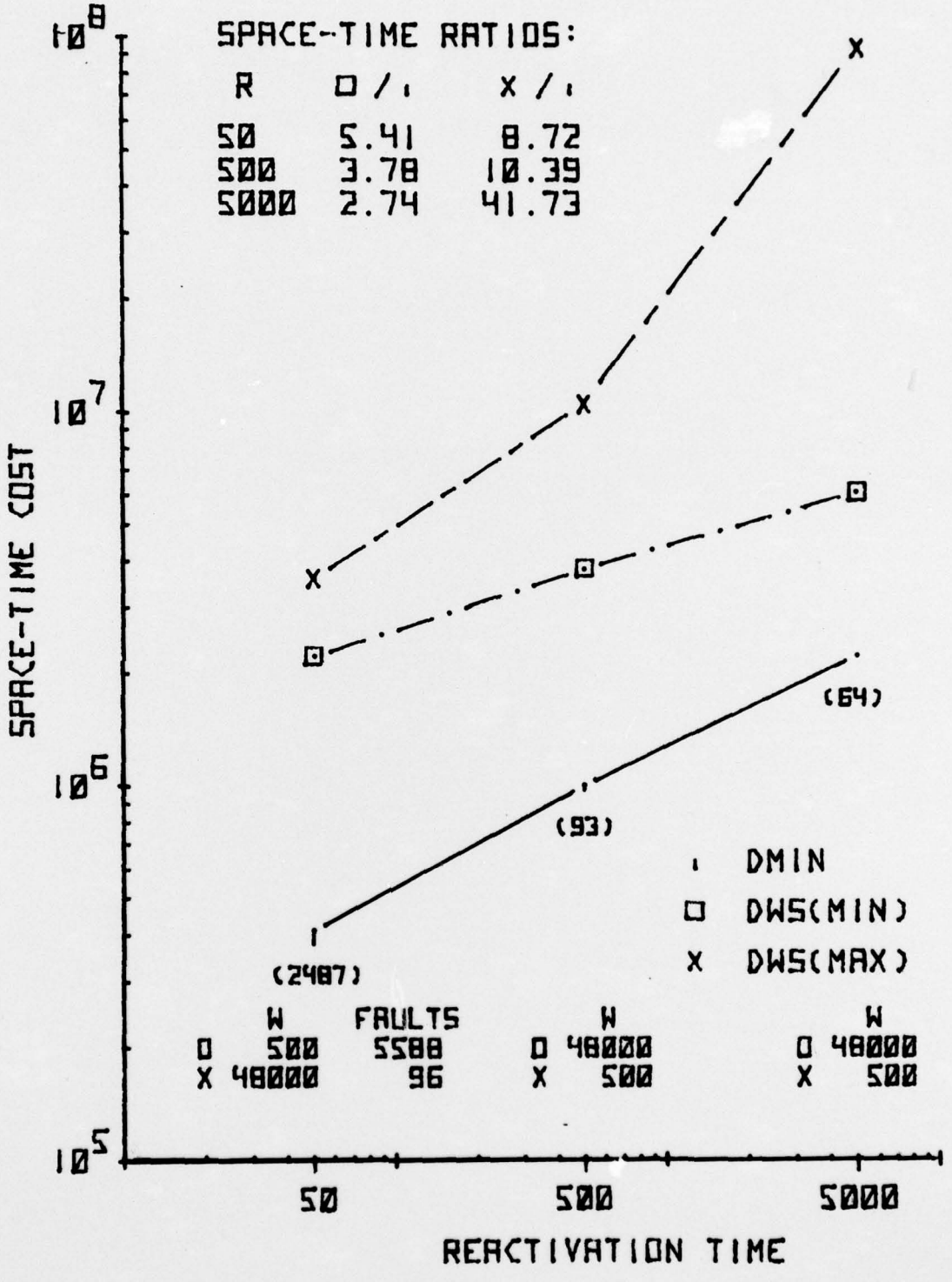


Figure 38. Comparison of DWS with DMIN for LIST with a 512 byte page (space-time cost normalized to 4096 byte page).

would be degraded because the vacate/occupy decision would become more ambiguous. However, we did not expect such a large degradation in performance. It seems that the 512 byte page for LIST has the effect of making the fact that a page has not been referenced for a certain length of time a bad predictor of future reference patterns.

In Table 4, we have listed several representative values of the results of the DWS experiments for LIST for a 512 byte page. This table has the same format as Table 3. We see that for the DWS(MIN) values, there is little variation in the space-time cost ratio as we near the best W value for a given reactivation time. For example, for R equals 500, we see almost no change for the ratio as W varies from 32,000 to 48,000 memory references. This indicates that we had come close to the best possible space-time cost for DWS.

The results for Denning's algorithm are contradictory. For LIST with a 4096 byte page, we see that Denning's algorithm come very close to being an ideal heuristic dynamic allocation algorithm. For LIST with a 512 byte page, we see that Denning's algorithm performed very poorly--much worse than MIN or PFF. We cannot conjecture about which performance is typical. Future workers will hopefully answer this question with other experiments and will develop improved, more stable heuristics for dynamic allocation.

Table 4

Experimental Results of the DWS Algorithm for LIST with a 512 Byte Page vs DMIN

			SPACE-TIME RATIO: DWS/DMIN		
W	MULT	Faults	R: 50	500	5000
500	1	5588	5.41	10.39	41.73
1000	1	3939	5.66	8.96	33.95
2000	1	2762	6.40	8.23	28.93
8000	1	575	7.38	4.32	7.77
16000	1	292	7.79	4.04	5.06
32000	1	113	8.55	3.78	2.94
48000	1	96	8.72	3.78	2.74

	REACTIVATION TIME		
DMIN ALLOCATION	50	500	5000
SPACE-TIME COST	411136.9 376865.6	1003217.3 972638.8	2227356.8 2218896.8
FAULTS	2487	93	64

CHAPTER 5

CONCLUSION

We have described DMIN, an algorithm for computing a dynamic allocation of primary memory which minimizes the space-time cost of primary memory used for a program run. We have proven that the set of vacated intervals from the DMIN allocation may be placed in a stack whose depth is decreased as the reactivation time increases. We have described a method of implementation which we used for our simulations. We have proven the worst case complexity for our reduction algorithm. We have presented a method for complexity reduction which leads to fairly tight bounds above and below the optimal space-time cost.

Some recently published work [16] develops an algorithm, VMIN, similar in purpose to DMIN. VMIN, however, ignores the fact that the number of pages resident when a page fault occurs is variable. Our G-graph could be modified to implement VMIN by deleting all edges. These edges are, however, of critical importance for evaluating dynamic allocation when the allocation size varies widely as in Figure 24; i.e., precisely when dynamic allocation varies significantly from static allocation. We therefore feel that the additional complexity of DMIN is essential for appropriately evaluating dynamic allocation in regions where it is preferable to static allocation.

In our simulations, we compared DMIN allocations with MIN algorithm allocations. On the basis of our comparison we conclude that a dynamic allocation strategy is generally superior to a static allocation

strategy. The dynamic strategy has the biggest improvement in performance for the smallest reactivation times and the smallest page sizes.

From our comparisons of the DMIN allocations with the Page Fault Frequency algorithm and the Damped Working Set algorithm, we conclude that further work in the area of dynamic heuristic allocation algorithms is justified. For the PFF algorithm, the space-time cost performance is sporadic compared to DMIN. Also, the space-time cost is sensitive to the value of P, the page fault frequency parameter. For the DWS algorithm we observed widely varying performance. In one case Denning's algorithm had near ideal performance. In the other case, Denning's algorithm had the worst performance of any allocation algorithm used in our experiments. For the experiments performed, Denning's algorithm did give superior performance compared to the other versions of the Damped Working Set algorithm.

When optimal allocations are required for a range of reactivation time, the best method is to compute the optimal allocation for the smallest reactivation time first. Substantial processing time can be saved for successively larger reactivation times by using Theorem 2.5.1. For each new (larger) reactivation time, the optimal allocation need not be determined from the entire G_A-graph, but rather can be derived from the minimum subgraph of the previous (next smallest) reactivation time.

From our experiments with the RED algorithm, we suggest that future investigators implement a RED(2) algorithm. We conjecture that RED(2) would tend to converge more quickly for our implementation, since more nodes would be removed in a dual scan. All nodes removed by RED(1)

would be removed. Some pairs of nodes removed by RED(1) in two dual scans would be removed by one dual scan of RED(2). Furthermore, any two nodes whose sum of weights is less than or equal to R would be removed by RED(2).

There is also possible improvement in the maximum flow algorithm. We suggest that some work be done in developing heuristics to improve the method for an initial flow in the flow graph. Another possibility is to determine when it is better to find maximum flow for the $\overline{G}$-graph rather than the G-graph used here.

Another topic would be to study the effect of attaching a system cost to a page fault in addition to the space-time cost of the primary memory. This could be accomplished by biasing the occupy/vacate decision. For the DMIN algorithm, if space-time cost was not increased by vacating an interval, the interval was vacated. This decision could be biased by requiring a minimum saving of space-time cost before an interval is assigned to the vacated state. This strategy would save some page faults and hence time at the cost of a small increase in space.

Another interesting topic is to study the effect of using a distribution to generate a random variable for the reactivation time of each potential page fault. This would provide a better model, but one which is more expensive to implement. It would be useful to know if the constant reactivation time assumption provides a satisfactory approximation to the more realistic model.

Future research should include a more system-oriented approach to memory allocation rather than the simple program-oriented approach taken

here. The components of R could be treated explicitly with estimates of bottlenecks in secondary memory bandwidth, primary memory size, processor wait, etc. A more complete set of parameters could be defined. Analytic functions for their values in terms of system activity could be found. The vacate/occupy decision and a decision to add or shed job load could then be made as a function of system activity at the moment. The appropriate number of processors and the amount of memory space they require for system balance could also be studied.

Particularly as the more complex models are undertaken, efficient computed and bounded approximation techniques for finding the minimum space-time cost allocation would be highly desirable.

REFERENCES

1. L. A. Belady and C. J. Kuehner, "Dynamic space sharing in computer systems," Comm. ACM 12, 5 (May 1969), 282-288.

2. E. G. Coffman and P. J. Denning, Operating Systems Theory, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

3. L. A. Belady, "A study of replacement algorithms for a virtual storage computer," IBM Sys. J. 9, (1966), 78-101.

4. P. J. Denning, "The working set model for program behavior," Comm. ACM 11 (May 1968), 323-333.

5. A. J. Smith, "A modified working set paging algorithm," IEEE TC, Vol. C-25, 9 (Sept. 1976), 907-914.

6. W. W. Chu and H. Opderbeck, "The page fault frequency algorithm," FJCC (1972), 41 part I, 597-609.

7. T. C. Hu, Integer Programming and Network Flows, Addison-Wesley, Reading, Massachusetts, 1969.

8. L. R. Ford, Jr. and D. R. Fulkerson, Flows in Networks, Princeton University Press, Princeton, New Jersey, 1962.

9. E. A. Dinic, "Algorithm for the solution of a problem of maximum flow in a network with power estimation," Soviet Math. Dokl. 11 (1970), 1270-1280.

10. J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," J. ACM 19, 2 (April 1972), 248-264.

11. A. V. Karzanov, "Determining the maximal flow in a network by the method of preflows," Soviet Math. Dokl. 15 (1974), 434-437.

12. S. Even and R. E. Tarjan, "Network flow and testing graph connectivity," SIAM J. Comp. 4 (Dec. 1975), 507-518.

13. R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," IBM Sys. J. 9, (1970), 78-111.

14. K. R. Kaplan and R. O. Winder, "Cache-based computer systems," Computer, (March 1973), 30-36.

15. H. S. Stone, "Multiprocessor scheduling with the aid of network flow," University of Massachusetts report ECE-CS-75-7 (October 1975).

16. B. G. Prieve and R. S. Fabry, "VMIN - An optimal variable-space page replacement algorithm," Comm. ACM 19 (May 1976), 295-297.

VITA

Robert Lucius Budzinski was born in Chicago, Illinois, on May 14, 1950. He received the B.S. degree in Electrical Engineering in 1972 and the M.S. degree in Electrical Engineering in 1974, both from the University of Illinois at Urbana-Champaign. He was a research assistant in the Department of Computer Science from 1972 to 1974. He was a research assistant at the Coordinated Science Laboratory from 1974 to 1976.